dTEE: A Declarative Approach to Secure IoT Applications using TrustZone

Anonymous Author(s)

ABSTRACT

Internet of Things (IoT) applications have recently been widely used in safety-critical scenarios. To prevent sensitive information leaks, IoT device vendors provide hardware-assisted protections, called Trusted Execution Environments (TEEs), like ARM TrustZone. Programming a TEE-based application requires separate code for two components, significantly slowing down the development process. Existing solutions tackle this issue by automatic code partition while not successfully applying it in two complicated scenarios: adding trusted logic and interactions with secure peripherals.

We propose dTEE, a declarative approach to secure IoT applications based on TrustZone. dTEE proposes a rapid approach that enables developers to declare tiered-sensitive variables and functions of existing applications. Besides, dTEE automatically transforms device drivers into trusted ones. We evaluate dTEE on four real-world IoT applications and seven micro-benchmarks. Results show that dTEE achieves high expressiveness for supporting 50% more applications than existing approaches and reduces 90% of the lines of code against handcrafted development.

KEYWORDS

Internet of Things; ARM TrustZone; Declarative Language

1 INTRODUCTION

Nowadays, Internet of Things (IoT) applications are widely adopted for people to interact with physical environments. Such usages, especially under safety-critical scenarios (e.g., structure monitoring [41] and healthcare [5, 29, 33]), also raise serious security and privacy concerns [1, 2]. This safety problem is worsened by the vulnerable wireless connection and limited computation resources of IoT devices.

As a countermeasure, IoT device vendors leverage hardware-assisted Trusted Execution Environments (TEEs) such as ARM TrustZone [4] and Intel SGX [18] to enhance the ability of IoT devices against attacks. TEE separates an application into two parts: a trusted application (TA) and a client application (CA). Security-sensitive functions and variables reside in TA, and the application logic in CA can interact with TA only using TEE APIs.

Unfortunately, hardening an existing non-secure IoT application with TEE is not easy. First, the separated architecture of TEE-based applications forces developers to carefully design the *control-flow* between TA and CA, which needs an in-depth knowledge of TEE APIs. Second, developers should also take care of the *data-flow* to prevent the disclosure of security-sensitive variables.

To help developers port existing applications to secure ones, researchers introduce automatic approaches [24, 35] to encapsulate sensitive data into TA and generate glue codes between TA and CA. Such approaches are suitable for generating simple trusted applications, but fail when facing the following two complicated scenarios: (1) *Complicated trusted logic.* As the complexity of trusted IoT application grows, developers may not be satisfied with only protecting the data but also try to add customized logic such as encryption. (2) *Secure peripheral interactions.* The most noticeable distinction between IoT and desktop applications is the interaction of sensing and actuating peripherals. However, how to automatically convert the software libraries containing peripheral interactions with TEE is left unsolved.

In order to address the above challenges, we present **dTEE**, a declarative approach to secure IoT applications using TrustZone. Compared with existing approaches, dTEE has three distinct advantages. First, dTEE provides a novel *declarative programming model*, D-lang, to facilitate the expression of complicated trusted logic and simplify the development by our well-designed primitives. Second, dTEE proposes a *peripheral-oriented library porting mechanism* to automatically port the IoT applications with peripheral access. Third, to minimize the switching overhead between CA and TA, dTEE further generates the trusted applications using *dCFG-based code partitioning*.

To use dTEE, developers first specify their trusted application logic and the target hardware platform in a declarative manner using D-lang. Afterward, the dTEE system takes the D-lang and the non-secure application source code as input, automatically partitions the original code into CA and TA side, and optimizes the control-flow to improve the performance of the generated application. Finally, dTEE outputs the compiled CA and TA according to the specified platform, and developers can deploy the application without any modification.

We implement dTEE system on top of a widely-used TEE in IoT applications, ARM TrustZone, and evaluate its performance with four representative TEE-based applications [25–27, 36] and seven micro-benchmarks. Results show that: (1) D-lang can reduce 90.03% of the lines of code against handcrafted development. (2) Compared with existing approaches, dTEE supports automatic security enforcement for 50% more applications. (3) Our dCFG-based code generation technology could improve at most 1.1x the runtime overhead of the generated application. (4) dTEE incurs less than 6.6% overhead in terms of execution time for IoT applications compared to the manual approach, which is acceptable.

We summarize the contributions as follows:

- We present dTEE, a novel system to accelerate the development of trusted IoT applications using a declarative approach.
- To maximize the efficiency of the TEE-based applications, we formulate the code generation problem as a graph optimization problem and leverage an efficient solver to solve it.
- We implement dTEE and extensively evaluate its expressiveness, efficiency, and overhead. Results show that dTEE

can significantly reduce the additional efforts to develop a secure IoT application.

The rest of this paper is organized as follows: Section 2 introduces related work of dTEE. Section 3 describes its background and programming model. Section 4 presents usage example of dTEE. Section 5 and Section 6 describe the design and implementation details. Section 7 evaluates the expressiveness, performance and overhead. In addition, Section 8 discusses the dTEE and future work. Finally, section 9 concludes the paper.

2 RELATED WORK

Automatic code partitioning for TEE-based applications. Glamdring [24] is a source code partitioning framework built for C applications of Intel SGX. It allows developers to specify the data needed to be protected by code annotation. This framework uses automatic static program analysis to find the program dependencies of annotated data, which the pre-built partition specification would filter. Consequently, a source-to-source transformation compiler separates one C application into two parts (i.e., Untrusted code in the REE and Trusted code in the TEE). Another related work is the automatic partitioning of Android applications for TrustZone [35]. Similar to the [24], it uses taint analysis to find the related candidate statements of annotated sensitive data. The native Java App is automatically separated into two components: 1) privileged code with TEE-specific commands wrapper using JNI, transformed to a TA manually later, and 2) normal code compiled to a CA. Occlum [37] is a library operating system (LibOS) for Intel SGX while supporting both security and efficiency. Using Occlum, the original use cases can automatically execute in the SGX without partitioning into two versions manually. The Occlum addresses inter-process and intra-enclave isolation using a novel software instrumentation technique.

While much of the research into automatic code partitioning has focused on TEE-based desktop applications, an equally important yet underexplored problem is how to rapidly develop complicated trusted IoT applications. The previous works cannot satisfy the vigorous applications of IoT since they have a wide variety of peripheral interactions.

TrustZone-based applications. Nowadays, much research focuses on enhancing the security of IoT applications. In this paper, we use the following four existing works to illustrate how users develop trusted IoT applications with dTEE.

DarkneTZ [26] is designed to protect sensitive layers of the DNN model using TrustZone, against privacy attacks (e.g., Membership Inference Attacks). This work proposed a TEE-based neural network framework for training and inferring DNN models based on porting subset codes of the Darknet [34] framework for execution in the secure world. For specifically, the partition points of the DNN layers marked by users indicate the execution environment when calculating forward and backward.

MQT-TZ [36] is a lightweight middleware using MQTT protocol on top of TrustZone. Compared with the original MQTT service (e.g., mosquitto[8]), this middleware attempt to deploy the MQTT broker into the secure world. In this work, brokers permanently store the unique symmetric key of each subscriber in the secure world, and re-encryption operates before the broker receives the encrypted messages from publishers and forwards them to the subscribers. Since finding the unique key of subscribers through the secure storage mechanism has underestimated overhead, MQT-TZ proposed an LRU Cache in the TEE.

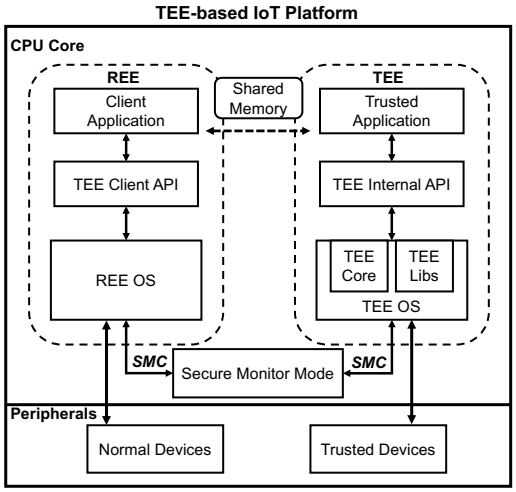


Figure 1: Programming model of TEE-based applications.

Alidrone [25], a reliable system that enables drones to prove the alibi of No-Fly-Zones. Benefits from the privilege, the drone operator has no permission to temper the geographic information. The GPS sensor driver is implemented in the TEE kernel space, exposing an entry to the TEE user space. To prove the data information is from a unique drone, the plain tuple (i.e., latitude, longitude, and timestamp) sampled by the secure driver is signed by a private key (e.g., RSA private key). For confidentiality, with a symmetric key (e.g., 128 bits AES key) in the TEE, the authentic GPS tuples are encrypted to a cyphered text.

TZ4Fabric [27] is designed to execute the smart contracts of Hyperledger Fabric (HF) in TEE instead REE. HF is an open-source shared ledger with permission blockchain. A smart contract is called chaincode in HF. TZ4Fabric protects the chaincode part of HF with the TEE and remains the proxy part in the non-secure world.

3 BACKGROUND

3.1 Programming Model of TEE-based Apps

For developers, building a TEE-based application is not effortless. Typically, developers produce a CA and a TA for one security demand from scratch in two execution environments. Figure 1 shows the programming model of TEE-based applications. On the REE (Rich Execution Environment) side, a CA is a normal application in the user space of REE, with the TEE Client APIs to communicate with TA. The only way for CA to switch REE to the TEE is to change the CPU mode to secure monitor mode by invoking the SMC (Secure Monitor Call). On the TEE side, a TA is executing in the user space of TEE with the TEE Internal APIs to interact with the TEE OS, which is built with TEE OS core and functional libraries. In various TEE OSs (OP-TEE [23], QSEE [32], Trusty [3], etc.), TEE Client APIs and TEE Internal APIs are uniformed by the GlobalPlatform [14].

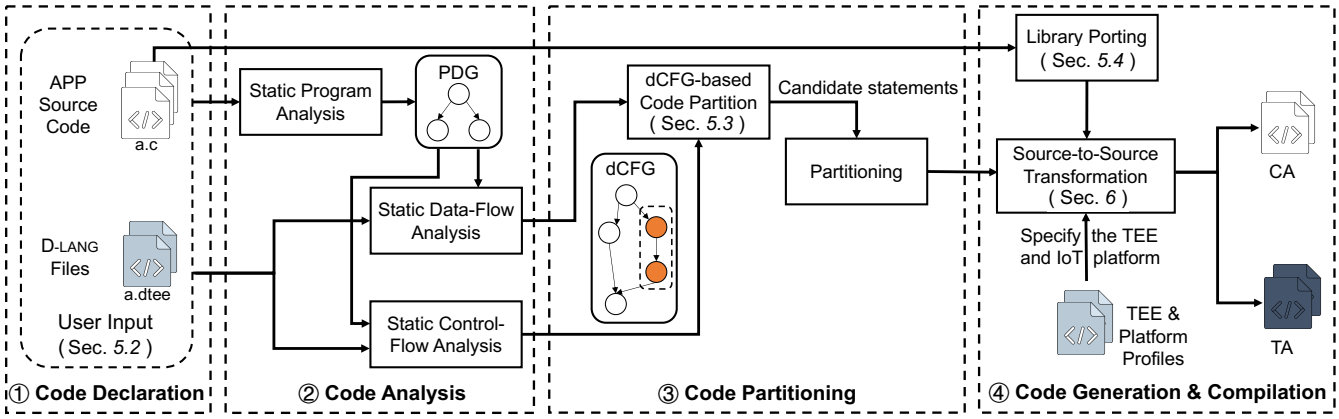


Figure 2: Illustration of dTEE workflow.

Nevertheless, the workflow is more complex for existing applications if developers try to port them to secure ones based on TEE. First, the developers must define where are the sources and sinks of sensitive information. Second, they need to find as many corresponding statements as possible to their best efforts. Then, they must manually use a mount of TEE Client APIs and TEE Internal APIs, such as `TEEC_InitializeContext()`. They need to use the TEE Client APIs and TEE Internal APIs carefully for communicating between the CA and TA, e.g., allocating shared memory, passing, and checking the transmitted parameters. Besides, they must remove or port the existing functions not supported in the TEE. Finally, the two applications can be successfully cross-compiled under their security demands.

3.2 Secure Peripheral Accessing of TEE Apps

Unfortunately, IoT developers need more struggle to port their existing applications to secure ones. A normal desktop application usually has no interactions with peripherals, while the IoT application has numerous. In the secure scenario, developers can configure controllers on the TEE-based IoT platforms to set some peripherals as trusted devices. Thus, the CA cannot access the trusted peripherals. Porting a peripheral-oriented application into TEE is complicated because the user space of TEE in which the TA is running has no permission to access trusted peripherals. To tackle this, developers must declare unique physical registers address in the kernel of TEE about their own IoT platform, following the datasheets. In addition, developers need to rewrite their peripheral interaction APIs because the TEE's fundamental read and write operations differ from the REE.

4 DTEE USAGE

In this section, we first introduce the design ethos and workflow of dTEE. Then, we propose real-world usage cases of D-LANG.

4.1 Goals

Soundness. Our primary goal is to attain soundness, a formidable endeavor given the intricate interconnections between the REE and TEE. For instance, meticulous sanitization of the transformation interfaces is imperative to ensure seamless handling of pointers [31, 39].

Application independent. Our second goal is to achieve application independence. Specifically, we hope that D-LANG programs should be developed independently that not require source code modifications to existing applications. Hence, the developers only need to use simple statements to declare sensitive data and functions of secure demands and do not need to make significant efforts to manually find the dependency and transform the original code to TEE-based code.

Ease of programming. Our third goal is to allow developers to program trusted applications easily. Hence, we propose a declarative language on top of the well-known SQL language for developers to specify their trusted requirements in their familiar way. We further enhance the declarative language with the support of native C code and security-related APIs for developers to express complex trusted logic. Declarative languages are well known for their ability to express complicated operations with short programs. The customized declarative language we developed will be detail described in Section 5.2.

Efficiency. The last goal revolves around augmenting the execution efficiency of applications while minimizing the associated overhead resulting from the enhancement of application security.

```
1 /* DroneApp/main.c */
2 int main() {
3     /* ... */
4     rawDataType rawData = getRawData();
5     GPSDataType gpsData = parseRawData(rawData);
6     /* ... */
7 }
```

Figure 3: The core application logic of DroneAPP.

Other examples

To see how we can secure existing applications using dTEE, let us consider an example shown in Figure x.

With dTEE, the Alidrone developers with the original insecure version can secure the application using the following four stages.

① **Code declaration.** dTEE must know which data or functions users want to protect in the TEE. The developers provide declaration

```
/* Case1/drone.dtee */
FROM DroneAPP/main.c FUNC main {
    TZ_DATA_BACKWARD gpsData;
}
```

Figure 4: D-LANG code snippet of Case 1.

```
/* Case2/drone.dtee */
FROM DroneAPP/main.c FUNC main {
    TZ_DATA_BACKWARD gpsData;
    INSERT_AFTER 5
        STRING_TZ *private_key;
        TZ_genKey(RSA_KEYPAIR, 2048, private_key, NULL);
        char *signed_gpsData;
        TZ_sign_rsa_sha1(signed_gpsData, private_key, &gpsData, ...);
    END_INSERT
}
```

Figure 5: D-LANG code snippet of Case 2.

files that program using our D-LANG (see Section 5.2). This stage is the only stage needed users to do.

In many realistic scenarios, users have different requirements on sensitive statements based on the security development demands. We can see from the following cases how dTEE can satisfy users' diverse security requirements, including Alidrone.

Case 1: Users require that the integrity of GPS data must be protected by TEE. They can declare the variables to be protected with keyword `TZ_DATA_BACKWARD` to cope with this requirement, as Figure 4 shows. The "backward" means dTEE should find the code slice which influences the GPS data in a backward direction, e.g., the `rawData` and `getRawData()`.

Case 2: Further, similar to Case 1, Alidrone wants to sign the GPS data in the TEE. This requires users to create a new variable `signed_gpsData` which still resides in the normal world but adds new secure programmable logic, i.e., signing the GPS data, in the TEE. Users can write the declarative statements of Figure 5 to achieve the core implementation of Alidrone. To insert new code logic, we use `INSERT_AFTER` and `END_INSERT` to declare an insertion block. The `private_key` is declared with `STRING_TZ`, specifying it is a variable in the TEE. The `signed_gpsData` (line 7 of Figure 5) is declared by the `char` keyword of Standard C, and thus this statement is inserted into the REE while the remaining statements are inserted into the TEE. The `TZ_genKey()` and `TZ_sign_rsa_sha1()` are built-in functions provided by the dTEE to facilitate developers.

② **Code analysis**. This stage finds all the corresponding sensitive statements of declared data and functions to satisfy users' secure demands, protecting sensitive items' integrity or confidentiality. Based on the application source code, dTEE uses static program analysis to generate a PDG (Program Dependency Graph) containing data and control dependency. After users define the sensitive data in the D-LANG files, dTEE regards them as tainted sources or sinks, protecting their confidentiality and integrity, respectively. Then, the user input and PDG are analyzed with data-flow and control-flow to search for tainted statements that will be partitioned to the TEE.

③ **dCFG-based code partition.** The code partition is responsible for generating the optimal partition of the result in the analysis phase. The dTEE leverages the dCFG(declarative Control-Flow Graph)-based code partitioning mechanism to improve the efficiency of the final CA and TA, detailed in Section 5.3.

④ **Code generation and compilation.** The last stage is responsible for generating source code and compiling CA and TA. The library porting mechanism is built to automatically port the existing sensitive drivers of the original application to the TEE-based ones. We will further give a detailed description in Section 5.4. The source-to-source transformation leverages the TEE Client APIs and TEE Internal APIs to set up routine communications between CA and TA, detailed in Section 6.1.

5 DESIGN OF DTEE

5.1 Threat Model

dTEE aims to relieve developers' efforts to secure existing IoT applications without source-level modifications. dTEE secures the applications without violating the principle of the developer's demands. More specifically, dTEE only follows the guidance of developers' annotations. For example, suppose the existing application has a code logic of accessing data D, and developers want to permanently store D in the TEE using the keyword `TZ_STORE` (described in Section 5.2). In addition to securing D in the TEE, dTEE still exposes the read, write and delete APIs of D to the REE because developers do not declare protecting the confidentiality of D. Thus, we emphasize that the CA may still has the ability to access sensitive data in the TA, which depends on the developers' demands.

In terms of software level, we consider a powerful attacker with the ability to access all platforms supporting TEE physically. The commodity OS of the platform in the REE is untrusted and could be compromised by adversaries. We rely on the protection mechanisms offered by the TEE to shield sensitive data and code. In addition, we assume that the TEE OSs (e.g., OP-TEE) are trusted and reliable. We consider that the user's driver is invulnerable because we focus on converting existing applications to TEE-based under the guidance of developers.

In terms of hardware level, we assume that the SoC of developers is trusted, and other components outside of SoC are assumed to be vulnerable, including peripherals. We rely on the hardware protection offered by the TEE device vendors.

5.2 Declarative Development Language

To address the limited expressiveness problem of existing TEE-development approaches [24, 35], we propose a declarative approach, D-LANG, for developers to specify their application logic of the trusted IoT application. Table 1 shows the keywords of D-LANG. Compared to existing works, D-LANG has two distinct features that improve expressiveness.

The following two critical issues with previous research have not yet been solved: (1) To transform the existing application to the TEE-based one, developers may need to add customized logic of new security demands. (2) The protection granularity of statements is not satisfied for development. To this end, D-LANG is designed for developers that add *new trusted logic* to complicated applications

Table 1: D-LANG keywords overview

Statement types			Keywords	Comments
General			FROM, FUNC	The general format for D-LANG files. FROM{...} FUNC{...}
Declarative Development of Trusted Logic	Configuration		@DRIVER, INSERT_AFTER, END_INSERT	Configure the TEE environment.
	Variable declaration		INT_TZ, CHAR_TZ, FLOAT_TZ, STRING_TZ, STRUCT_TZ	Declare secure variables which have not existed in the original apps.
	Built-in functions	Cryptography	TZ_genKey(), TZ_sign_rsa_sha1(), ...	Built-in crypto functions, including enc/dec, sign/verify, hash, and random.
		Peripherals	TZ_digitalWrite(), TZ_digitalRead(), ...	Built-in peripheral APIs.
Tiered Protection	Variables	Permanence	TZ_STORE	Permanently protect data based on the secure storage mechanism.
		Temporariness	TZ_DATA_ONLY, TZ_DATA_BACKWARD, TZ_DATA_FORWARD, TZ_DATA_ALL	Tiered degrees for temporary protection.
	Functions	Protection	TZ_FUNC	Protect functions.
		Substitutions	TZ_FUNC_SUB	Substitute an original function with a new function.

and provides *tiered protection* of variables and functions that is more fine-grained than previous works.

Formally, a D-LANG program consists of three types of statements: general statements, trusted logic statements, and tiered protection statements. The general statements are to locate the original file that has security demands.

Trusted logic statements. These statements are designed to add new trusted logic, which was neglected by previous works. The `@DRIVER` and the pair of `INSERT` are to enrich the functionality in TEE.

The existing work focuses on desktop applications but neglects the demands of peripherals. Hence, the most crucial expressiveness of D-LANG is the interaction of physical devices. We emphasize that the most noticeable distinction between IoT and desktop applications is the interaction of sensing and actuating peripherals, which is why the previous cannot work.

To ameliorate this issue, D-LANG is designed with the `@DRIVER` keyword, which specifies the particular profile of the hardware platform (see details in Section 6.2). The non-secure device drivers of IoT applications that we supported can automatically transform into identical secure driver functions.

```
1 @DRIVER "profile.h";
```

Another crucial issue that previous work still has not addressed is the insertion of the new trust code. As the example of Section 4 shows, the `INSERT_AFTER` (line 4 of Figure 5) and `END_INSERT` (line 9 of Figure 5) are a pair of configuration statements for inserting arbitrary new program logic that is not in the original code.

The variable declaration statements are built for developers that add trusted variables in the TEE for new secure demands, while the existing approaches are missing this vital feature. Since numerous IoT applications are not developing based on TEE, these statements can generate new secure variables in the TEE that are absent in original codes for new development demands. The declarations are functionally identical to standard C. Note that these variables are defined in the TEE. We design five popular types of variables, while the variables that are defined by Standard C remain in the REE. For example, as (line 5 of Figure 5) shows, the `private_key` is declared with `STRING_TZ`, specifying it is a variable in the TEE. The `signed_gpsData` (line 7 of Figure 5) is declared by the `char` keyword of Standard C, and thus this statement is inserted into the REE.

In addition, the users can use the built-in TEE-based functions to develop the TA rapidly. We provide the most operations in TEE-based IoT applications. For example, the `TZ_genKey()` (line 6 of Figure 5) function to generate a user-specified key and the `TZ_sign_rsa_sha1()` (line 8 of Figure 5) function to sign the data with the private key of RSA by the SHA-1 algorithm.

Further, we design various built-in peripheral functions for accessing the devices. The users can declare the insert keywords of configuration statements and use the homogeneous functions of `TZ_digitalWrite()` to develop their secure IoT application drivers rapidly.

Tiered Protection. Existing works [24, 35] can automatically protect the variables. However, they do not provide a programming interface that could enable users to specify to what extent they want to protect the variables, e.g., only protect the integrity or protect both integrity and confidentiality. Hence, dTEE provides tiered protection of variables and functions, as shown in Table 1.

We discover that non-tiered annotation of sensitive data does not satisfy IoT development demands. For example, in the autopilot scenario, where safety is a critical topic, the central controller desires the vehicle to be controllable and thus means that the velocity decision is trusted. Further, the users cannot tolerate the geolocation information leaking. Towards the aforementioned goals, the developers need to protect not only the *integrity* of vehicle steering but also the *confidentiality* of the position.

Therefore, we propose five tiered degrees of sensitive variables for protection in TEE, one for permanent and four for temporary. (1) `TZ_DATA_STORE`. This keyword permanently protects sensitive data based on the secure storage mechanism [17] of TEE. This mechanism is used to protect crucial sensitive data, such as biological information. The following keywords support temporary tiered protection. (2) `TZ_DATA_ONLY`. This keyword only protects the annotated data, excluding tainted statements. (3) `TZ_DATA_FORWARD`. This keyword protects the declared data and other tainted data and functions that forward the data flow for confidentiality. (4) `TZ_DATA_BACKWARD`. As shown in Figure 4, contrary to the previous, this keyword protects the sources of specific data through the backward data-flow and control-flow for integrity. (5) `TZ_DATA_ALL`.

This combined TZ_DATA_BACKWARD and TZ_DATA_FORWARD for both confidentiality and integrity.

```
FROM DroneAPP/main.c FUNC main {
// To permanently store the data in secure storage
    TZ_STORE gpsData;
}
```

However, it is not expressive enough for D-LANG to only provide the insert statements. The TZ_FUNC_SUB keyword is another usage dimension of trusted logic. This keyword can substitute the original function with a new function implemented in D-LANG files that facilitate users modifying their customized logic of a specific function.

```
int encrypt_sub(/*...*/) {
    // substitution implementation
}
FROM DroneApp/main.c FUNC main {
    TZ_FUNC_SUB parseRawData ((/**/) encrypt_sub(/**/);
}
```

Figure 6: Illustration of logic-based program declaration of D-LANG.

For example, as shown in Figure 6, the parseRawData() function is a non-TEE-based function in the original code of Alidrone (Figure 3), while its functionality cannot satisfy the new security demands. Hence, users can write the TZ_FUNC_SUB keyword to substitute parseRawData() by the encrypt_sub() function.

5.3 dCFG-based Code Generation

As we described in Section 3, according to the programming model of TEE, TA is protected against unauthorized access from CA and other TAs. Thus, after the developer uses D-LANG to declare their demands of function protecting, dTEE automatically put these functions into TA to protect them.

However, the world switching overhead is a considerable proportion. Through the measurements on Raspberry Pi 3B+, we conduct that five times of world switch (Figure 7 (a)) could occur the execution overhead of ~250 microseconds. If these five calls could be packaged into one in the TA, like Figure 7 (b), the world switching overhead would be reduced to ~70 microseconds. Hence, we leverage a dCFG-based code partitioning mechanism to obtain the optimized code partition by considering the world switching overhead. This mechanism contains two stages. First, we construct a declarative CFG (dCFG) based on the basic CFG generated in the code analysis process. Second, we propose an algorithm to obtain an optimized partition with dCFG.

dCFG construction. Our goal is to package the statements that continuously access TA resources, and put as many statements into the package as possible without violating the principle of the developer's demands. To clarify function logic and variable dependency, we propose a declarative graph, dCFG.

In order to highlight the advantages of dCFG in analyzing code logic, we have adapted the Alidrone program in Figure 3 by adding more control statements to the original application, shown in Figure 9. The program is always in a while loop. When the flag is

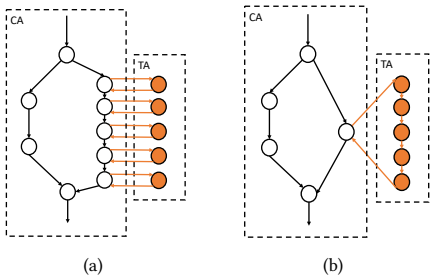


Figure 7: World switching cost optimization.

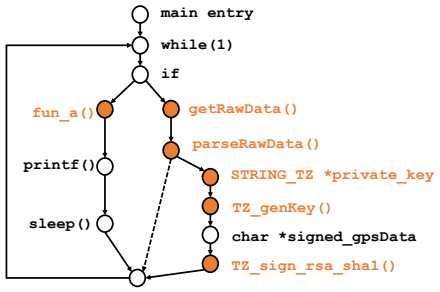


Figure 8: Generation of dCFG.

true, the GPS signal is collected. Otherwise, the secure function fun_a() is called and sleeps for 1000 ms. Now we consider the following secure demands: the developer not only wants to protect data collection in Figure 5 but also increases the demand for protection of the fun_a() function.

We have formulated the following rules to construct a dCFG according to the source code and D-LANG files mentioned earlier.

Step 1. We need to generate a DAG for each function according to the source code, which directly represents the program logic of CA. Each statement in the code represents a node, such as a variable declaration statement. Note that the caller function is regarded as a node, instead of the internal logical structure of the callee function.

Step 2. For the INSERT keywords, according to the code line number and code logic of INSERT, insert the corresponding node in the corresponding position of the DAG (e.g., the signed_gpsData node in the Figure 8), which indicates that CA called the TA function. dTEE marks the node according to the security attribute of the statement.

Step 3. For TZ_DATA, we get the statements related to the protected variable according to the taint analysis and mark them in the CFG diagram. If all statements in a function are tainted, the whole function is considered a protected function, which will be put in TA. These protected functions will be handled in the next step.

Step 4. For TZ_SUB and TZ_FUNC, after replacing function according to FUN_SUB, we mark the statement fun_a that calls this function in the CFG diagram according to the TZ_FUNC keyword.

We generate a dCFG for this example according to the *rules*, as Figure 8 shows. In step 1, we generate a DAG where each node represents a statement in the example. In step 2, we add the nodes generated between INSERT_AFTER and END_INSERT. We mark the STRING_TZ node, the TZ_genKey node, and the TZ_sign_rsa_sha1

```
/* DroneApp/main.c */
int main() {
    /* ... */
    while(1){
        if (!flag) {
            fun_a();
            printf("wait...");
            sleep(1000);
        }
        else {
            rawDataType rawData = getRawData();
            GPSDataType gpsData = parseRawData(rawData)
      ;
            /* ... */
        }
    }
}
```

Figure 9: The core application logic of an adapted DroneAPP.

node because they are protected. In step 3, the `rawData` node and the `gpsData` node will be marked according to the result of the taint analysis. In step 4, the `fun_a` node will be marked because it is a TA function call.

Partitioning algorithm. After completing the above steps, we get a specific graph consisting of statements, which cover all of the developer's demands.

The data flow graph can be represented by a directed acyclic graph (DAG) $G(V, E)$, where the vertices represent logic blocks and the edges represent data flow.

We formulate our partitioning problem as a numerical optimization problem. The optimal partitioning result can be viewed as assigning each logic block to its most appropriate location (i.e., either the secure world or the non-secure world) while ensuring security goals. We use a binary indicator X_{b_i,w_i} to denote the placement of the logic block.

$$X_{b_i w_i} = \begin{cases} 1 & \text{logic block } b_i \text{ is assigned to world } w_i \\ 0 & \text{logic block } b_i \text{ is not assigned to world } w_i \end{cases} \tag{1}$$

where w_i represents the possible placement device of logic block b_i.

We define the full path as the path from a source vertex to a sink vertex, denoted as p. We use $len(p)$, $\delta(p)$ and $P(G)$ to indicate the length of path p, the number of vertices in path p, and the set of all full paths in graph G. Our optimization goal is thus denoted as $minmax_{p\in P(G)} len(p)$. Moreover, due to the $len(p)$ is the sum of data processing and transmitting latency across all possible placements, our objective is formulated with the binary placement indicator $X_{b_i,w_i} \in \{0, 1\}$ as:

$$\begin{aligned} \arg\min_{X} \max_{p\in P(G)} & \sum_{i=1}^{\delta(p)} \sum_{w_i \in W_i} X_{b_i w_i} T^{Comp}_{b_i w_i} \\ & + \sum_{i=1}^{\delta(p)-1} \sum_{\substack{w_i \in W_i \\ w_{i'} \in W_{i'}}} X_{b_i w_i} X_{b_{i'} w_{i'}} T^{Trans}_{b_i w_i w_{i'}} \end{aligned} \tag{2}$$

subject to:

$$\sum_{i=1}^{\delta(p)} \sum_{w_i \in W_i} X_{b_i w_i} m_{b_i} \le M$$

where i, i' are the adjacent vertices in path p (i.e., i'=i+1). W_i denotes the possible placements (i.e., secure world or non-secure world) of the i-th logic block. $T^{Comp}_{b_i w_i}$ denotes the data processing time of the i-th logic block on placement w_i, and $T^{Trans}_{b_i w_i w_{i'}}$ represents data transmitting time between block b_i of placement w_i and block b'_i of placement w'_i. We assume that the data transmitting time is negligible if the two consecutive logic blocks are placed on the same device. Thus we have:

$$T^{Trans}_{b_i w_i w_{i'}} = \begin{cases} r_{ii'} t_{ii'k} & w_i \neq w_{i'} \\ 0 & w_i = w_{i'} \end{cases} \tag{3}$$

where $r_{ii'}$ denotes the data size being transmitted on edge (i, i'). $t_{ii'k}$ is a method-specific factor representing the time to transfer data via method k from block i to the next block i'. There are two main methods to transmit data between two worlds, i.e., register-based transmission and shared-memory-based transmission. Typically, the former is faster than the latter but has a limited number of parameters (e.g., four). We recognize the numbers and types of parameters to automatically select which transmission methods.

This optimization problem presents an additional constraint due to the limited secure memory available within the TrustZone. For example, the maximum secure memory allowed by OP-TEE is 8 MB. Thus, the total memory consumption $\sum_{i=1}^{\delta(p)} \sum_{w_i \in W_i} X_{b_i w_i} m_{b_i}$ of all logic blocks placed in the secure world must be less than the maximum memory capacity M of the secure world, where m_{b_i} represents the maximum memory size occupied by the respective logic block.

We propose Algorithm 1 to partition the dCFG. First, set all nodes in dCFG as unvisited. Then, during a depth-first search, for each unvisited secure node in the graph, if the in-degree and out-degree of the node are both 1, it will be added to the node set of $V[i]$ that needs to be packaged. When the whole map search is completed, the set of $V[i]$ is the node that needs to be packaged into the TA.

However, not all the statements could be packaged into TA. Considering the demands declared by developers strictly, we will not package non-secure elements together. In addition, if the statement contains the control statements (e.g., `if` and `for`), it will not be separated into the TA. Thus, when the program is expressed in the form of dCFG, only the node with one in-degree and one out-degree will be considered.

5.4 Peripheral-oriented Library Porting Mechanism

It is crucial for dTEE can transform non-secure library files to TEE-based since IoT applications usually have lots of libraries based on physical interaction for functionalities, such as r/w GPIO and sampling sensors.

Furthermore, we discover three types of non-secure libraries when protecting them into TEE-based. Table 2 illustrates the details of library types. We first propose a peripheral-oriented library porting Mechanism. Since existing works lack concern about peripheral interactions, there is no solution adaptable to IoT applications today.

Portable libraries. This type of libraries is the simplest type of protection. These library files are restricted to all the functions supported by the TEE and no Linux system calls (e.g., `open()`,

Algorithm 1 dCFG partitioning algorithm

Input: the set of statements from project S
Output: the set of packaged statement nodes $V[1...n]$

```
1: generate CFG from S
2: n := start of CFG; i := 0; flag := false
3: mark each node of CFG to be unvisited
4: DFS(n)
5: procedure DFS(n)
6:     mark n visited
7:     if n.outDegree = 1 and n.inDegree = 1 then
8:         if n is secure then
9:             V[i].add(n)
10:            flag := true
11:        else
12:            if flag = true then
13:                i := i + 1
14:                flag := false
15:            end if
16:        end if
17:    end if
18:    for each o ∈ n.outDegree do
19:        if o is unvisited then
20:            DFS(o)
21:        end if
22:    end for
23: end procedure
24: return V
```

`close()`, `write()`) invocation. The common TEEs usually support a subset of Standard C Library, and as such, portable libraries are constrained with strong-restrict conditions.

For example, the Quirc [6] is a widely used library for extracting and decoding QR codes in IoT scenarios. If the developers try to execute these algorithms in the TEE, they can use dTEE with two approaches. The first approach is that users provide the source codes of Quirc library and the header files. Then, the dTEE can cross-compile the Quirc source code to a static library (i.e., `libquirc.a`). Afterward, the static library is linked to the TA when the dTEE compiles the TA. Finally, the TA can use Quirc APIs provided by the header files in the TEE, achieving the developers' goals. The second approach is the users only provide a finished cross-compiled static library of Quirc. They need to declare the location and name of the `libquirc.a` file in the declarative file. Consequently, the dTEE can link this static library when compiling the TA. We emphasize that this is the only type of library that existing works can successfully protect in the TEE.

Auto-transformable libraries. This type of libraries is more complex than Portable libraries hence its interaction with the physical world using peripherals. If an IoT library uses `mmap()` operation of Linux to map physical memory addresses of peripherals (e.g., control registers of sensors) to virtual memory areas for accessing GPIO operation, the dTEE can automatically transform it to the TEE-based.

For instance, WiringPi [15] is a widely used C/C++ peripheral library built for the Raspberry Pi, a powerful and popular IoT platform. The accessing operation to the GPIO of Raspberry Pi is ultimately implemented by the `mmap()` function that the WiringPi library invokes. Note that almost all the GPIO functions implemented in WiringPi are based on the `mmap(` function, such as `digitalWrite()` and `digitalRead()` functions. Suppose we only duplicate codes of these functions into the TEE. However, it is a failure when trying to run the IoT applications because a TA is executing in the user space of TEE that has no permission to map physical addresses to the secure memory area. Only the kernel of TEE has the privilege to map. Consequently, dTEE provides pre-built functions in the TEE kernel for GPIO access. If users try to protect peripheral drivers implemented by WiringPi, they can use dTEE with two steps: 1) Providing a specific profile (see Section 6.2), and 2) Declaring the profile using configuration statements in Section 5.2. Then, the dTEE can use automatic substitute corresponding hardware-driven functions with the pre-built APIs. Finally, the sensitive drivers can successfully compile into a TA.

Manually-transformable libraries. This type of libraries is the most complex situation due to the program logic has changed. If developers have new program logic of secure demands on existing IoT applications, they can use D-lang to insert their new security programs. In addition, note that the existing non-secure which cannot smoothly port in the TEE (e.g., Linux system calls) or significantly increase the TCB (Trusted Computing Base) size (e.g., OpenSSL library [28]), dTEE can address these problems with the substitution and insertion statements in the D-lang.

The first example is MQT-TZ [36], which encrypts clients' messages by the software encryption of OpenSSL. Suppose developers want to encrypt the messages with a secure hardware chip acceleration. In this scenario, it is useless to protect the non-secure encryption operation in the TEE since the encryption logic has changed.

A further example, DarkneTZ [26] has changed the DNN inference logic of existing framework [34]. Specifically, DarkneTZ adds the new program logic by setting partition points that indicate the sensitive and normal layers execute in the TEE and REE, respectively. Hence, in this scenario, it is useless to follow the logic of the original codes like [24] and [35] since the developers have added new security logic.

To tackle the aforementioned problems, the developers can use dTEE with the following solutions. 1) Users could specify the library of cryptography to use, such as `Libmbedtls` and `LibTomCrypt`, or hardware accelerating chips. 2) Users could use insert statements that add their new trusted logic.

6 SYSTEM IMPLEMENTATION

In this section, we will describe the implementation details and the portability of dTEE.

6.1 Implementation details of dTEE

The code analysis of dTEE is built as an extension to the Frama-C [9], an open-source, extensible analysis framework for C software. We implement the source-to-source transformation of code generation in Java. The transformation can generate the TEE Client APIs

Table 2: Three types for protecting libraries of IoT apps

Types	Portable libraries	Auto-transformable libraries	Manually-transformable libraries
Program logic	No changed	No changed	Changed
Transformation	No need	Need	Need
Conditions	All the functions supported by TEE, and no operations of Linux system calls.	The peripheral libs of IoT apps use mmap() operation to access GPIO.	The developers add new logic, or the libs increase the TCB significantly.
Examples	`Libquirc` [6], `Libnmea` [19]	`LibwiringPi` [15]	`Libopenssl` [28], `Libcrypto` [28]
Konstantin et al. [35]	✓	✘	✘
Glamdring [24]	✓	✘	✘
dTEE (Our paper)	✓	✓	✓

and TEE Internal APIs in the CA and TA, respectively. These generated wrappers are used for initialization, opening, communication, closing, and finalization between the CA and TA.

Analysis and partition. We extract the semantics of D-LANG files by the keywords, which facilitate static program analysis. For the tiered sensitive data annotated, we use the "Impact" and "Scope & Data-flow browsing" plug-ins [10, 11] of Frama-C for forward and backward flow. In addition, we build call graphs of the source code and traverse all reachable statements from these entry points. For partition, based on the "Slicing" plug-in of Frama-C [12], we build specifications to filter the candidate-tainted statements. We then use the dCFG-based optimization to reduce the overhead.

Auxiliary code. After the code partition stage of dTEE, the source code is separated into two parts but is still inadequate. The separated code cannot maintain basic REE and TEE sessions. Thus, dTEE needs to generate the auxiliary code. To achieve TEE independence for dTEE, we use the standard TEE Client APIs and TEE Internal APIs that are unified by the GlobalPlatform [14] and deployed in the popular TEEs, e.g., OP-TEE [23], QSEE [32], and Trusty [3].

Table 3 shows the main APIs that dTEE generated. Typically, the CA and TA consist of five skeleton functions to maintain a connection between CA and TA. First, we generate the pair of initialization, session, and finalization code in the CA and TA. Then according to the function's invocation that CA called TA, we specify the invocation commands. However, due to the programming model of TEE-based, the data types are strictly defined. Thus, dTEE reviews the transferred data flow for security and relies on the constant data types. In addition, each TA that resides in the TEE has a unique identifier, which we need to random generate.

On the non-secure world side of the application, the auxiliary codes of normal function are generated based on the parameters received from the caller and the returns. However, the number of arguments passed between CA and TA is limited to four [], which is unsuitable for the requirements of most functions. In detail, we ameliorate this issue as follows. If the number of parameters is less than four and their types are integer or string, dTEE can use the native APIs (e.g., `TEEC_VALUE_INPUT` and `TEEC_MEMREF_TEMP_INPUT`) to pass. Otherwise, dTEE allocates appropriate shared memory to serialize the parameters and deserialize in TA afterward. A more complex issue is the pointers in C applications. A straightforward way is that, if the CA function passes a pointer, we generate a temporary variable that dereferences to the pointer and passes it to the TA. Then, on the TA side, we also generate another temporary variable that references the former. However, Pointers could be poisoned by the adversary to mount confused-deputy attacks, arbitrarily nested pointer-to-pointer style, and have the vulnerability of TOCTOU (Time of Check, Time of Use) attacks. Thus, we exerted considerable effort to meticulously inspect and sanitize the pointers. For example, to prevent confused-deputy attacks, we encapsulate the dereferenced pointers in the TEE within objects that manage their lifetime and access. Thus, they can restrain direct access to the underlying memory. As a result, we address the pointer issue without destroying the existing function logic. For example, suppose the parameters passed to the TA from CA are a pointer to a structure containing an integer and a string. In that case, we first dereference the structure pointer and allocate fixed bytes of shared memory. Then, we serialize the integer and string values into the shared memory. After that, the pointer values are transmitted to the TEE side successfully. To avoid changing the programming logic in the TA, we construct an identical structure and pointer in the TA according to the structure variable in the CA. Thus, if we deserialize the passed parameter and re-reference, the remains code of TA can still execute without modifications.

On the secure world side of the application, the transformation not only generates the wrappers of parameters and commands but also tackles the drivers of trusted devices. In specific, dTEE provides uniform APIs of heterogeneous hardware platforms. Hence, these APIs build the built-in peripheral functions and the sensitive drivers declared trusted. The built-in functions are independent of the TEE OS. For example, on the Raspberry Pi 3B+ platform, the built-in function `TZ_digitalWrite()` tries to set the pin as "output" mode according to the addresses of GPFSEL, GPSET, and GPCLAR registers [7]. The logic of built-in functions is hardware-dependent but TEE-independent. But, the generated code of built-in functions is TEE-dependent. For example, in one of the TEEs, OP-TEE, we implement the built-in peripheral code in the `optee_os/core/arch/arm/pta` for pseudo-TAs [38].

6.2 Porting dTEE to different platforms

Different hardware platform leverage different programming APIs to switch between CA and TA or perform peripheral accessing. The generated code of dTEE should conform to the APIs specification of the corresponding platform.

Currently, we implement dTEE on the Raspberry Pi platform with ARM TrustZone technology. However, it is worth noting that the implementation of dTEE is not tightly coupled with a specific hardware platform. In this subsection, we describe how we make dTEE portable among different platforms and TEE technologies.

Table 3: The main TEE Client APIs and TEE Internal APIs generated by dTEE for auxiliary code.

Category	TEE Client API	TEE Internal API	Description
Skeleton Functions	`TEEC_InitializeContext()`	`TA_CreateEntryPoint()`	It called only once to construct a TA.
	`TEEC_OpenSession()`	`TA_OpenSessionEntryPoint()`	To open a session with CA and TA that connects the REE and TEE.
	`TEEC_InvokeCommand()`	`TA_InvokeCommandEntryPoint()`	To invoke a TA command by the CA. The invocation commands are specified by the CA and TA.
	`TEEC_CloseSession()`	`TA_CloseSessionEntryPoint()`	To close an opened session and disconnect the REE and TEE.
	`TEEC_FinalizeContext()`	`TA_DestroyEntryPoint()`	It is the last command to destruct a TA.
Transferred Data Types	`TEEC_VALUE_INPUT`, `TEEC_VALUE_OUTPUT`, `TEEC_VALUE_INOUT`	`TEE_PARAM_TYPE_VALUE_INPUT`, `TEE_PARAM_TYPE_VALUE_OUTPUT`, `TEE_PARAM_TYPE_VALUE_INOUT`	These types indicate that the data is an integer. INPUT indicates the transferred data is REE to TEE, while OUTPUT indicates the direction is TEE to REE, and INOUT means bi-direction.
	`TEEC_MEMREF_TEMP_INPUT`, `TEEC_MEMREF_PARTIAL_INPUT`, `TEEC_MEMREF_TEMP_OUTPUT`, `TEEC_MEMREF_PARTIAL_OUTPUT`, `TEEC_MEMREF_TEMP_INOUT`, `TEEC_MEMREF_PARTIAL_INOUT`	`TEE_PARAM_TYPE_MEMREF_INPUT`, `TEE_PARAM_TYPE_MEMREF_OUTPUT`, `TEE_PARAM_TYPE_MEMREF_INOUT`	These types indicate that the data is allocated to shared memory. The TEMP specifies that the memory reference is temporary, and PARTIAL specifies that the memory is registered in the partial region of its parent shared memory block.
Identifier	`TEEC_UUID`	`TEE_UUID`	The UUID type is defined in RFC4122 [20], which values are used to identify TAs. We use a UUID generator for each TA.

Portability among different TEE technologies. There exist several TEE implementations for different hardware vendors. Furthermore, the TEE implementation may differ even for the same vendor. Hence, to enhance the portability among different TEE technologies, dTEE leverages a *TEE portability profile* to decouple the TEE technology and the code generation. Each TEE profile contains adequate information that is needed to generate code on for different TEE. According to our investigation, we summarize the following three kinds of information required in the TEE profile.

The first and foremost substance in the TEE profile is the specification of programming model and APIs for different TEEs. For example, the SGX leverages `ECALL` and `OCALL` to switch between CA and TA, while the TrustZone application needs first open a secure session and then uses `TEEC_InvokeCommand` to call the methods residing in TA. The second is the file structure and header files because different TEE has its own header files which consist of its built-in APIs. The third difference is the application generation tool-chains. For example, the generation process of building an SGX-based application and the TrustZone-based application is different. To compile a TrustZone application, dTEE may directly compile the CA and TA parts of the application. However, compiling an SGX-based application requires first compiling the CA, then generating the trusted code using `sgx_edger8r` tool, compiling TA to the dynamic library, and signing it using `sgx_sing`. Hence, to bridge this gap, dTEE defines application generation script for different TEEs as part of the TEE profile.

Portability among different platforms. The above TEE portability profiles capture the differences among TEE vendors. Nevertheless, within the same TEE vendors, the generated TEE code may differ because of the hardware platform portability. For example, both Raspberry Pi and Xilinx Ultra96 v2 development board use ARM TrustZone. However, due to the different memory layouts and pin mapping of these boards, the generated code differs.

Hence, to ease the process of porting dTEE to different hardware platforms, we also introduce a *platform portability profile* that captures the platform heterogeneity. The platform portability profile consists of two parts.

Firstly, the profile includes a well-defined Makefile for each platform because different platforms have different cross-compilation toolchains. Secondly, the hardware peripheral accessing APIs of each platform are different. IoT development boards usually use reserved memory regions to map the I/O for developers to access the peripherals. However, because different boards have different memory layouts and pin-out, so the peripheral accessing APIs are also different. Hence, our profile also incorporates the memory layout of each board, and the code generation module of dTEE leverages this layout to handle the peripheral-related code automatically.

7 EVALUATION

In this section, we evaluate dTEE to answer the following three questions: (1) Does dTEE achieve better expressiveness and rapid development than existing approaches at IoT applications? (2) What is the dCFG-based optimization improvement performance of dTEE? (3) What is the overhead of dTEE?

In our experiments, we explore the capabilities of a open source TEE, OP-TEE [23], implemented on a widely used IoT platform, Raspberry Pi 3B+ board [30]. We installed Raspbian Kernel Version 4.14.98 as REE OS and OP-TEE Version 3.4.0 as the TEE OS.

7.1 Case Studies

To highlight the expressiveness of dTEE and D-lang, we show that they can cover the core functionality of four representative TEE-based IoT applications. The details of applications are introduced in Section 2. For MQT-TZ [36], TZ4Fabric [27], and DarkneTZ [26], we implement their core functionality while leaving out less relevant details. For Alidrone [25], due to the lack of source code, we implement it by the main idea and techniques in the literature. We

Table 4: Real-world applications and micro-benchmarks implemented by dTEE

App	Orig. app LOC	D-LANG LOC	D-LANG Keywords	REE &TEE Switch Numbers	Orig. app Binary Size (B)	CA Binary Size (B)	TA Binary Size (B)	Konstantin et al. [35]	Glamdring [24]	**dTEE**
Print	5	3	3	2	7.8K	12.6K	98K	✓	✓	✓
cJSON [13]	19	1	1	2	31K	12.4K	112.4K	✘	✓	✓
Concat	33	3	3	2	7.9K	6.1K	100K	✓	✓	✓
Blink [15]	21	1	1	2	8.3K	12.7K	97.7K	✘	✘	✓
Temp [15]	181	4	4	4	8.5K	12.7K	98K	✘	✘	✓
Humi [15]	181	4	4	4	8.5K	12.7K	98K	✘	✘	✓
Alidrone [25]	129	11	9	2	29K	12.6K	121.4K	✘	✘	✓
MQT-TZ [36]	291	5	5	2	13K	12.6K	100K	✘	✘	✓
TZ4Fabric [27]	89	8	6	2	8.5K	12.7K	108.2K	✘	✓	✓
DarkneTZ [26]	33.5K	31	31	4	366K	384K	112K	✘	✘	✓
Socket [22]	118	N/A	N/A	N/A	34K	N/A	N/A	✘	✘	✘

implement the original version for each application and the secure version using the D-LANG declaration for comparison.

MQT-TZ. MQT-TZ is a lightweight middleware attempt to deploy the MQTT broker into TEE. We implemented the original version of MQT-TZ with `libopenssl`. Specifically, we combined the `save_key` and `aes` applications in the MQT-TZ brokers as the benchmark application. The program logic is that the broker maintains the key of each client, and encrypts the data that will transfer to the clients. There is a `struct` of `mqttz_client` has four members: `cli_id` (a client ID), `iv` (an initial vector of the symmetric key), `key` (a symmetric key) and `data` (payload). We use 128 bits of AES key to encrypt the data. To secure the source application of MQT-TZ, as Figure 10 shows, we use a total of six keywords: two to locate the sensitive data, two to permanently store the `iv` and `key`, and two to substitute the encryption function with a built-in function `TZ_enc_aes_cbc()`.

```
1 int encrypt_sub(unsigned char *plain_text, int
        plain_text_len, unsigned char *key_id, unsigned
        char *iv_id, unsigned char *cipher_text, int
        key_size) {
2     return TZ_enc_aes_cbc(plain_text, plain_text_len,
        key_id, iv_id, cipher_text, key_size);
3 }
4 FROM mqt-tz/hot_cache/main.c FUNC main {
5     TZ_STORE dest->iv;
6     TZ_STORE dest->key;
7     TZ_FUNC_SUB encrypt(/**/) encrypt_sub(/**/);
8 }
```

Figure 10: Expressing MQT-TZ with D-LANG.

In MQT-TZ, the client value of the AES `key` and `iv` should be permanently stored in the secure storage of the TEE. Thus, we use statements of `TZ_STORE` on the two variables to implement the storage. For this particular example, the original non-secure version use `encrypt()` which is implemented by the `libopenssl`. To secure the encryption operation, MQT-TZ needs to implement the identical `encrypt()` in the TEE, whereas in D-LANG, a total of 3 lines of code is needed. The key reason for the reduction is that the patterns of cryptography in the TEE are recognized as templates of dTEE that provide built-in functions.

TZ4Fabric. TZ4Fabric is designed to execute smart contracts in TEE. In our implementation of the original application, we focus on the three smart contract functions: `create()`, `add()`, and `query()`. To demonstrate the identical security effect as TZ4Fabric, we use D-LANG to declare the three functions mentioned above using the `TZ_FUNC` keyword, as shown in Figure 11. The `TZ_FUNC` keyword can also secure *subfunctions*. For example, in the `create()` function, there are three subfunctions (i.e., `create_get_state()`, `create_put_state()`, and `create_write_response()`) could be called. With the call graphs, dTEE selected the related statements and separated them into the secure world. As a result, the three operations of smart contracts are secure in the TEE.

```
1 FROM chaincode_proxy/main.c FUNC fabric {
2     TZ_FUNC create();
3     TZ_FUNC add();
4     TZ_FUNC query();
5 }
```

Figure 11: Expressing TZ4Fabric with D-LANG.

DarkneTZ. DarkneTZ is designed to secure sensitive layers of the DNN model in TEE to prevent privacy attacks. In our implementation, we focus on the inferencing of DNN. The implementation of DarkneTZ takes 31 D-LANG keywords, as shown in Figure 12. To specific, we use a number of 20 `TZ_FUNC` keywords to declare sensitive data (e.g., `make_softmax_layer`), two pairs of `INSERT_AFTER` and `END_INSERT` keywords to add trust logic in the functions of `network_predict()` and `load_weights_upto()`. Compared to the original version, the new security logic added is designed for executing the non-sensitive layers of the neural network in the non-secure world and the sensitive layers of the neural network in the secure world. Thus, we use the pair of `INSERT_AFTER` and `END_INSERT` keywords in the loading weight stage (i.e., `load_weights_upto()`) and the prediction stage (i.e., `network_predict()`) of the inference to add the new logic. Specifically, we need to insert the new logic of loading network weights from REE and TEE, respectively, and separate the last five layers into the TEE. Hence, compared with previous work, our improved method of adding new programming logic is very useful in one of our case studies.

```
FROM darknet/main.c FUNC main {
    TZ_FUNC make_softmax_layer();
    /* ... */
    // We omit the other 20 functions that need to be
       protected.
}
FROM darknet/src/network.c FUNC network_predict {
    INSERT_AFTER 713
    /* ... */
    END_INSERT
}
FROM darknet/src/parser.c FUNC load_weights_upto {
    INSERT_AFTER 1611
    /* ... */
    END_INSERT
}
```

Figure 12: Expressing DarkneTZ with D-lang.

Alidrone. We introduced two usage examples in Section 4, which are simplified cases. To represent Alidrone, one of the realistic applications with peripherals, we sufficiently exploit the D-lang expressiveness. First, the original version uses a GPS peripheral with `libnmea` [19]. Second, it uses a cryptography operation that we need to separate into the TEE.

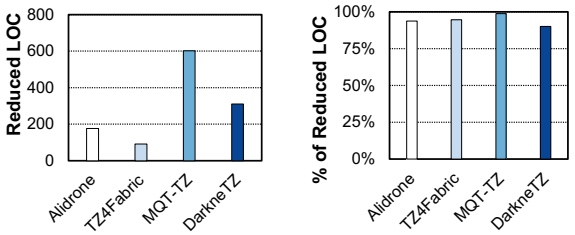
As a countermeasure, we provided a profile of the hardware platform (i.e., Raspberry Pi 3B+) to indicate the physical addresses of GPIO that the dTEE could auto-transform the libraries into the TEE. As Figure 13 shows, we use `@DRIVER` to specify the profile path. To transform the signature operation in the original version, we use the tiered-protection keywords to secure the data and substitute-function keywords to secure the function. For example, we use `TZ_DATA_ONLY` to secure the constant of the private key of RSA as a TEE runtime variable, while for the GPS data, we use `TZ_DATA_FORWARD` to protect the subsequent operations.

```
@DRIVER "profile.h"
void EvpSign_sub(const unsigned char* in, int in_size,
       unsigned char* sign) {
    TZ_sign_rsa_sha1(sign, in, Privatekey2048_E,
       Privatekey2048_D, Privatekey2048_N, &gpsData,
       2048);
}
FROM drone.c FUNC main() {
    TZ_DATA_ONLY Privatekey2048_E;
    TZ_DATA_ONLY Privatekey2048_D;
    TZ_DATA_ONLY Privatekey2048_N;
    TZ_DATA_FORWARD data;
    TZ_FUNC_SUB EvpSign() EvpSign_sub();
}
```

Figure 13: Expressing Alidrone with D-lang.

7.2 Expressiveness and Lines-of Code Reduction

Benchmarks. To be comprehensive, we use two sets of benchmarks implemented by dTEE: Real-world applications and micro-benchmarks. The real-world applications illustrate how dTEE expresses the existing IoT apps, including those we used as related works. For micro-benchmarks, to evaluate the expressiveness and rapid development of dTEE, on the one hand, we select typical IoT tiny examples from WiringPi [15] that contains LED blinks, reading temperature, and humidity; on the other hand, we use a small library cJSON [13] and two tiny function apps (i.e., Print and Concat). The Print app is for printing a string and the Concat app is for concatenating strings. In addition, we select a normal Socket app [22] for sending messages by networks.



(a) Reduced LOC. (b) Percentage of reduced LOC.

Figure 14: Reduced LOC and code rate by dTEE.

Results of benchmarks. Table 4 shows the results of benchmarks and Figure 14 shows the reduced LOC of real-world applications. Specifically, observations are as follows:

(1) On the expressiveness, as Table 4 shows, we can observe that 90.9% of applications can be implemented by dTEE, especially the IoT applications that existing works are not adapted. Due to the peripheral-oriented library porting mechanism (see Section 5.4), dTEE can provide automatic transformation of the peripheral drivers of various IoT devices to the trusted ones. However, dTEE cannot adapt to protecting socket functions since importing the network stack will remarkably increase the attack surface of TCB [21, 40].

(2) On the LOC reduction, as shown in Figure 14, we find that dTEE can reduce more than 90% lines of code (LOC) in real-world IoT applications. Since D-lang has a lot of built-in functions of peripherals and cryptography, dTEE achieves few keywords and LOC of D-lang to represent the security demands of trusted IoT applications, shown in Table 4.

7.3 Overhead of dTEE

We evaluate the overhead of dTEE from two aspects, the static memory overhead and the execution time overhead on benchmarks. The results exclude the DarkneTZ because it lacks the manual approach to transformation. Since [24] is based on Intel SGX applications and [35] is designed for Android applications, we do not compare the performance of dTEE and them.

Table 4 shows the binary size of CA and TA. Compared with original applications, the CA incurs little memory overhead, while TA has a large binary size because TA has more TEE internal functions and is linked with static libraries. We emphasize that only built-in functions of D-lang will produce a few additional memory sizes compared with the manual approach.

Figure 15 shows the complete execution time of benchmarks, and Figure 16 shows the execution time of invoking the core function command (CMD), i.e., `TEEC_InvokeCommand()` of benchmarks. The dTEE has comparable performance compared to the manual approach and only incurs up to 6.6% of the execution overhead compared to the manual partitioning approach. For the micro-benchmarks, the execution time of the manual and our automatic approach is almost identical. The reason is that the glued code generated by dTEE is almost comparable to the manual method. For complex real-world applications, we discover that the execution

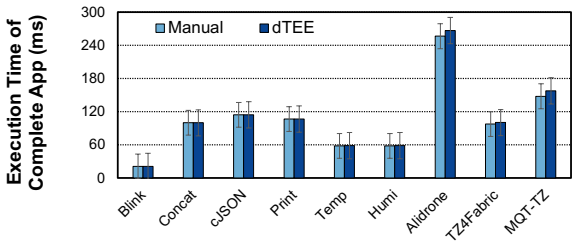


Figure 15: Complete execution time of TEE-based applications. The development approaches of manual and dTEE are compared.

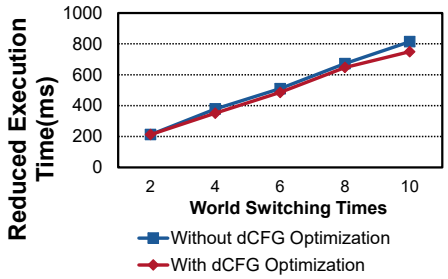


Figure 17: Execution time of different world switching times with/without dCFG.

time of applications generated by the dTEE is slightly slower than the manual development. That is because the auxiliary code for gluing the REE and TEE generated by dTEE is generic, which could be redundant compared to the manual approach. For example, in the MQT-TZ, when encrypting multiple data to the same client, the manual approach could cache the keys read from the secure storage instead of re-generating a read API for each encryption as dTEE does. This issue remains because the glued code generated by dTEE is designed to be generic. However, it could still be solved by setting rules to identify and remove redundant code automatically.

7.4 Performance Optimization

We demonstrate the improved performance of dCFG-based code partition (Section 5.3) on the Alidrone app. In specific, Alidrone has two operations, digesting with the SHA-1 algorithm and signing with an RSA private key. To optimize these operations, dTEE generates an entry function that invokes them in the TEE instead of invoking them from REE individually.

As Figure 17 shows, with the switching times of TEE and REE increasing, the overhead (e.g., changing privileged mode and preserving CPU context) of execution time increases, but dCFG-based code partition can improve performance by about 1.04x~1.1x.

8 DISCUSSION

We summarized three types of libraries to tackle the challenge of automatically porting an IoT library into a TEE-based one. The peripheral-oriented library porting mechanism (Section 5.4) can secure the peripheral drivers using `mmap()` of Linux, especially for the GPIO libraries. In our prototype, for the more complex drivers (e.g., a camera driver using DMA memory or SPI bus), developers cannot use dTEE to transform them automatically. However, they could write D-LANG code to substitute or insert the logic of drivers manually, which incurs considerable effort. The literature [16] proposed

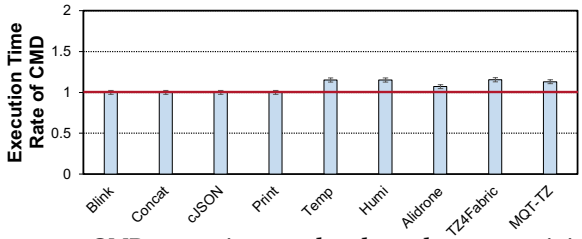


Figure 16: CMD execution overhead rate between original application and dTEE applications. The red line (—) represents the manual approach.

minimum viable drivers for ARM TrustZone. dTEE is compatible with the drivers in [16], and as such, the built-in functions could be supplemented for automatically transforming more rich hardware drivers.

In the future, we will consider introducing automation for identifying sensitive data without users manually declaring, which can further relieve the developers' effort. In addition, we will consider designing more application-specific operators to improve performance.

9 CONCLUSION

We present dTEE, a declarative approach to secure IoT applications based on TrustZone. With dTEE, users could declare tiered-sensitive variables and add new trusted logic for their security demands. Compared with existing works, dTEE achieves high expressiveness for supporting 50% more applications. We evaluate dTEE on real-world IoT applications and seven micro-benchmarks. Results show that dTEE reduces 90% of the LOC against handcrafted development.

REFERENCES

[1] Mohammed Al-Khafajiy, Safa Otoum, Thar Baker, Muhammad Asim, Zakaria Maamar, Moayad Aloqaily, Mark Taylor, and Martin Randles. 2021. Intelligent control and security of fog resources in healthcare systems via a cognitive fog model. *ACM Transactions on Internet Technology (TOIT)* 21, 3 (2021), 1–23.
[2] Tejasvi Alladi, Vinay Chamola, et al. 2020. HARCI: A Two-Way Authentication Protocol for Three Entity Healthcare IoT Networks. *IEEE Journal on Selected Areas in Communications* 39, 2 (2020), 361–369.
[3] Android. 2022. Trusty TEE. https://source.android.com/security/trusty. (2022).
[4] ARM. 2022. TrustZone for Cortex-A. https://www.arm.com/technologies/trustzone-for-cortex-a. (2022).
[5] Hasina Attaullah, Tehsin Kanwal, Adeel Anjum, Ghufran Ahmed, Suleman Khan, Danda B Rawat, and Rizwan Khan. 2021. Fuzzy-Logic-Based Privacy-Aware Dynamic Release of IoT-Enabled Healthcare Data. *IEEE Internet of Things Journal* 9, 6 (2021), 4411–4420.
[6] Daniel Beer. 2022. Quirc. https://github.com/dlbeer/quirc. (2022).
[7] BROADCOM. 2022. BCM2835 ARM Peripherals. https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf. (2022).
[8] The Eclipse Foundation. 2023. Eclipse Mosquitto - An open source MQTT broker. https://mosquitto.org. (2023).
[9] FRAMA-C. 2022. Frama-C - Framework for Modular Analysis of C programs. https://frama-c.com. (2022).
[10] FRAMA-C. 2022. Impact analysis plug-in. https://frama-c.com/fc-plugins/impact.html. (2022).
[11] FRAMA-C. 2022. Scope & Data-flow browsing plug-in. https://frama-c.com/fc-plugins/scope.html. (2022).
[12] FRAMA-C. 2022. Slicing plug-in. https://frama-c.com/fc-plugins/slicing.html. (2022).
[13] Dave Gamble. 2022. Ultralightweight JSON parser in ANSI C. https://github.com/DaveGamble/cJSON. (2022).
[14] GlobalPlatform. 2022. GlobalPlatform. https://globalplatform.org. (2022).
[15] Gordon. 2022. WiringPi. https://github.com/WiringPi/WiringPi. (2022).
[16] Liwei Guo and Felix Xiaozhu Lin. 2022. Minimum viable device drivers for ARM TrustZone. In *Proceedings of the Seventeenth European Conference on Computer*

Systems. 300–316.

[17] Shunrui Huang, Chuanchang Liu, and Zhiyuan Su. 2019. Secure Storage Model Based on TrustZone. In *IOP Conference Series: Materials Science and Engineering*, Vol. 490. IOP Publishing, 042035.

[18] Intel. 2022. Intel®SoftwareGuardExtensions (Intel®SGX). https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html. (2022).

[19] Jacketizer. 2022. libnmea: Lightweight C library for parsing NMEA 0183 sentences. https://github.com/jacketizer/libnmea. (2022).

[20] Paul Leach, Michael Mealling, and Rich Salz. 2005. *A universally unique identifier (uuid) urn namespace*. Technical Report.

[21] Seung-seob Lee, Hang Shi, Kun Tan, Yunxin Liu, SuKyoung Lee, and Yong Cui. 2019. S2Net: Preserving Privacy in Smart Home Routers. *IEEE Transactions on Dependable and Secure Computing* 18, 3 (2019), 1409–1424.

[22] Browny Lin. 2022. Simple socket example. https://gist.github.com/browny/5211329. (2022).

[23] Linaro. 2022. Open Portable Trusted Execution Environment. https://www.op-tee.org. (2022).

[24] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 285–298.

[25] Tianyuan Liu, Avesta Hojjati, Adam Bates, and Klara Nahrstedt. 2018. Alidrone: Enabling Trustworthy Proof-of-Alibi for Commercial Drone Compliance. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 841–852.

[26] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. 2020. DarkneTZ: Towards Model Privacy at the Edge using Trusted Execution Environments. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*. 161–174.

[27] Christina Müller, Marcus Brandenburger, Christian Cachin, Pascal Felber, Christian Göttel, and Valerio Schiavoni. 2020. TZ4Fabric: Executing Smart Contracts with ARM TrustZone:(Practical Experience Report). In *2020 International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 31–40.

[28] OpenSSL. 2022. OpenSSL. https://www.openssl.org. (2022).

[29] Nidhi Pathak, Anandarup Mukherjee, and Sudip Misra. 2020. Reconfigure and reuse: Interoperable wearables for healthcare IoT. In *2020 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 20–29.

[30] Raspberry Pi. 2022. Raspberry Pi 3 Model B+. https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus. (2022).

[31] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM computing surveys (CSUR)* 51, 6 (2019), 1–36.

[32] Qualcomm. 2022. Mobile Security Solutions. https://www.qualcomm.com/products/features/mobile-security-solutions. (2022).

[33] Mohsin Raza, Muhammad Awais, Nishant Singh, Muhammad Imran, and Sajjad Hussain. 2020. Intelligent IoT Framework for Indoor Healthcare Monitoring of Parkinson's Disease Patient. *IEEE Journal on Selected Areas in Communications* 39, 2 (2020), 593–602.

[34] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet. (2013–2016).

[35] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. 2016. Automated Partitioning of Android Applications for Trusted Execution Environments. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 923–934.

[36] Carlos Segarra, Ricard Delgado-Gonzalo, and Valerio Schiavoni. 2020. MQT-TZ: Hardening IoT Brokers Using ARM TrustZone:(Practical Experience Report). In *2020 International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 256–265.

[37] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 955–970.

[38] TrustedFirmware.org. 2023. Trusted Applications. https://optee.readthedocs.io/en/latest/architecture/trusted_applications.html. (2023).

[39] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. 2019. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1741–1758.

[40] Kailiang Ying, Amit Ahlawat, Bilal Alsharifi, Yuexin Jiang, Priyank Thavai, and Wenliang Du. 2018. Truz-Droid: Integrating TrustZone with Mobile Operating System. In *Proceedings of the 16th annual international conference on mobile systems, applications, and services*. 14–27.

[41] Wenjin Yu, Yuehua Liu, Tharam Dillon, Wenny Rahayu, and Fahed Mostafa. 2021. An integrated framework for health state monitoring in a smart factory employing IoT and big data techniques. *IEEE Internet of Things Journal* 9, 3 (2021), 2443–2454.