dTEE: A Declarative Approach to Secure IoT Applications Using TrustZone

Tong Sun[1], Borui Li[2], Yixiao Teng[1], Yi Gao[1], and Wei Dong[1]
[1]The State Key Laboratory of Blockchain and Data Security
College of Computer Science, Zhejiang University
[2]School of Computer Science and Engineering, Southeast University
{tongsun,tengyixiao,gaoyi,dongw}@zju.edu.cn,libr@seu.edu.cn

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 [52] and healthcare [10, 13, 36, 47]), also raise serious security and privacy concerns [5, 6]. 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 [8] and Intel SGX [23] 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 [29, 42] 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 applications 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 (declarative Control-Flow Graph)-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 the 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 the dTEE system on top of a widely-used TEE in IoT applications, ARM TrustZone, and evaluate its performance with four representative TEE-based applications [30, 33, 34, 43] 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 in our tested benchmarks. (3) Our dCFG-based code generation technology could improve at most 1.48x run-time overhead of the generated application compared to the non-optimization. (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:

979-8-3503-6201-5/24/$31.00 ©2024 IEEE
DOI 10.1109/IPSN61024.2024.00021

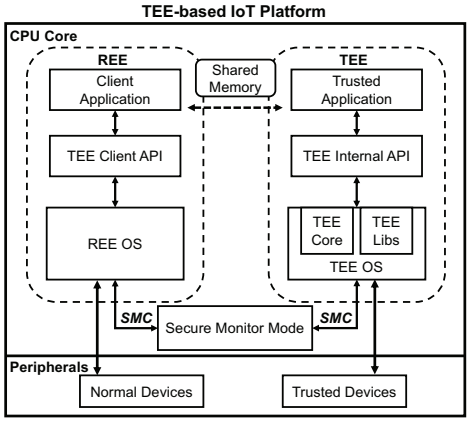

Figure 1: Programming model of TEE-based applications.

- We present dTEE, a novel system to accelerate the development of trusted IoT applications using a declarative approach. In addition, we propose the D-lang language to support expressive application development.
- 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.

2 BACKGROUND

2.1 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 rich execution environment (REE or non-secure world) side, a CA is a normal application that could use the TEE Client APIs to communicate with the 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 (or secure world) 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 OSes (e.g., OP-TEE [28], QSEE [39], Trusty [7], etc.), TEE Client APIs and TEE Internal APIs are uniformed by the GlobalPlatform [18].

Transitioning existing applications to TEE-based security necessitates a sophisticated workflow. Initially, developers must identify the sources and sinks of sensitive data. Subsequently, they should diligently map corresponding secure-sensitive statements to avoid leaking information. Communication between the CA and TA requires meticulous utilization of both TEE Client and Internal APIs, encompassing tasks like shared memory allocation and parameter transmission checks. Given that only a subset of POSIX interfaces are supported by the TEE OS, some features may need removal or modification. Only after these steps can the applications be compiled to meet security specifications.

2.2 Secure Peripheral Accessing of TEE Apps

Unfortunately, IoT developers need more struggle to port their existing applications to secure ones. In contrast to typical desktop applications which seldom interact with peripherals, IoT applications frequently do. Under secure conditions, TEE-based IoT platforms allow developers to designate certain peripherals as trusted, rendering them inaccessible to the CA. Transitioning a peripheral-centric application to TEE is intricate, given the TEE user space (where the TA operates) lacks permission to access these trusted peripherals. Addressing this requires developers to specify unique physical register addresses in the TEE kernel, aligning with platform-specific datasheets. Additionally, the peripheral can be set via TZASC [9] to be exclusively accessible by the TA, restricting access from the CA.

For example, to control an LED on a Raspberry Pi through TEE, developers must consult the datasheet for the GPIO address, then use TEE OS APIs like `GPFSEL` and `GPCLR` for basic operations. This requires a pseudo-TA approach [28], increasing development complexity. Furthermore, TEE's base operations, such as `read` and `write`, differ from REE, necessitating API adjustments. Hence, there is currently a lack of approaches to assist developers in swiftly transitioning IoT applications to TEE security.

3 RELATED WORK

Automatic code partitioning for TEE-based applications. Glamdring [29] 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 [42]. Similar to the [29], it uses taint analysis to find 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.

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.

Automatic code transformation for TEE-based applications. Twine [31] and WaTZ [32] are adaptations that integrate the WebAssembly micro runtime (WAMR) into TEEs; the former is predicated on the Intel SGX platform while the latter leverages ARM TrustZone. Their methodologies share similarities: a normal application is pre-compiled into WebAssembly (Wasm) bytecode using ahead-of-time (AOT) compilation, which is subsequently executed within the TEE through the runtime, encompassing the entirety of the application's bytecode. Occlum [44] is a library operating system (LibOS) for Intel SGX while supporting both security and efficiency. Using Occlum, the original App can automatically execute in the SGX without partitioning into two versions manually.

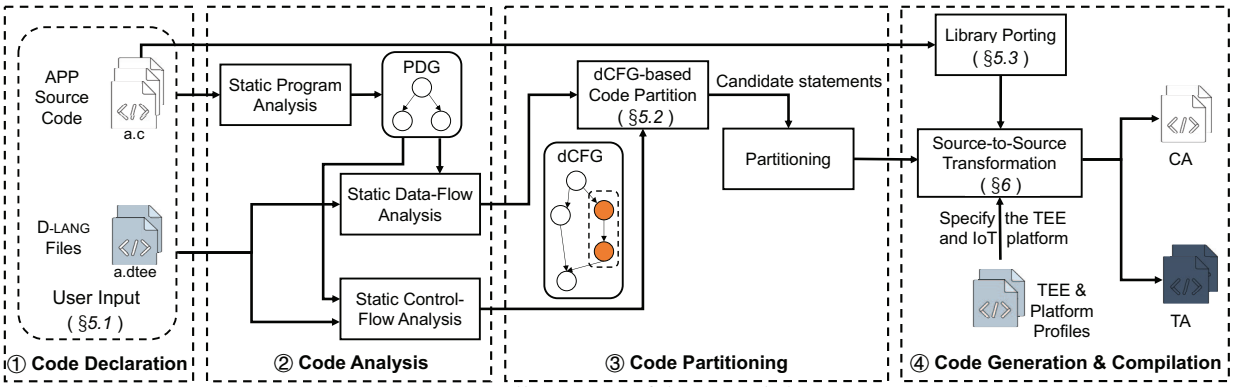

Figure 2: Illustration of dTEE workflow.

These efforts neglect the constrained nature of TEE secure memory, especially in TrustZone (typically under 20MB). Directly embedding a whole application into the TEE might surpass memory bounds. Thus, dTEE utilizes an auto-partitioning mechanism, inducting code segments into the TEE judiciously to prevent information breaches. Furthermore, these previous works are not feasible for IoT applications, whereas dTEE is adept at accommodating intricate IoT applications.

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 [33] utilizes TrustZone to shield critical DNN layers against privacy breaches like Membership Inference Attacks (MIA). It introduces a TEE-oriented neural network framework that ports selective components of Darknet [40] for secure world execution. Notably, to thwart MIA, it delegates deep layers (e.g., the last five) to TEE while the rest layers are operated in the REE.

MQT-TZ [43] is a lightweight TrustZone-based middleware employing the MQTT protocol. Contrasting traditional MQTT services (e.g., mosquitto[15]), it endeavors to secure the MQTT broker. Unique symmetric keys of subscribers are securely stored, with re-encryption preceding broker-handled encrypted messages. To minimize overhead in accessing keys through the secure storage mechanism, MQT-TZ introduces an LRU Cache within the TEE.

Alidrone [30] ensures drones can validate their compliance with No-Fly-Zones. Capitalizing on restricted privileges, drone operators cannot adulterate geographic data. The TEE kernel houses the GPS sensor driver, granting access to the TEE user space. To confirm unique drone-sourced data, secure driver-acquired tuples (latitude, longitude, timestamp) are signed using private keys (e.g., RSA). Ensuring confidentiality, tuples are encrypted using a symmetric key (e.g., 128-bit AES) within the TEE.

TZ4Fabric [34] enables Hyperledger Fabric (HF) smart contracts to run in TEEs, shifting from REE. HF, a permissioned open-source blockchain, refers to smart contracts as chaincode. TZ4Fabric safeguards the chaincode segment of HF in the TEE, leaving the proxy component non-secure.

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

To transform existing IoT applications into TEE-based secure applications, we present the following four objectives for dTEE.

Ease of programming. Our first goal is to allow developers to program trusted applications easily. Hence, we propose a declarative language D-LANG 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. We will describe D-LANG in §5.1.

Application independent. Our second goal is to achieve application independence. We aim for D-LANG programs to be developed such that existing applications need no source code modifications. Thus, developers can declare sensitive data and functions for security requirements through simple statements, eliminating the need for manual dependency identification and transformation to TEE-based code.

Efficiency. Our third goal revolves around augmenting the execution efficiency of applications while minimizing the associated overhead resulting from the enhancement of application security. A recent approach is to execute the entire application within the TEE [32], but it is constrained by the limited secure memory. We will show our novel partition design in §5.2.

Soundness. The last goal is to attain soundness, a formidable endeavor given the intricate interconnections between the REE and TEE. For example, meticulous sanitization of the transformation interfaces is imperative to ensure seamless handling of pointers [38, 49]. The implementation details are illustrated in §6.

4.2 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, i.e., dTEE tries to follow the guidance of developers' annotations.

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

```
/* DroneApp/main.c */
int main() {
  /* ... */
  rawDataType rawData = getRawData();
  GPSDataType gpsData = parseRawData(rawData);
  /* ... */
}
```

Listing 1: The core application logic of DroneAPP [30].

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

Listing 2: D-LANG code snippet of Case 1.

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 (System on Chip) 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. We do not consider side-channel attacks on the TEE, i.e., we assume that the TEE can protect the confidentiality and integrity of the program and data inside it.

4.3 Usage Example

Figure 2 illustrates the workflow of dTEE. To demonstrate how we can secure existing applications via dTEE, let us first show a typical drone application, as shown in Listing 1. Its primary functionality involves collecting raw data and converting it to a standardized GPS format. In this case, the application does not utilize a TEE for security. Alidrone [30] protects the variables and functions in lines 4 and 5 of Listing 1 to execute them within TEE, ensuring the collection of trusted GPS data. However, we cannot assume that all developers are experts in TEE programming like [30]. Now, we demonstrate how dTEE enables developers to seamlessly transition from this original version to a TEE-secured variant through a four-stage process, effectively reproducing Alidrone's modifications to the Listing 1 code.

① **Code declaration.** dTEE must know which data or functions that users want to protect in the TEE. Developers need to provide the declaration file that program using our D-LANG (see §5.1). 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 next show how dTEE can satisfy users' diverse security requirements in the following cases.

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_INTG` to cope with this requirement, as shown in Listing 2. The "INTEGRITY" means dTEE should find the code slice that influences the GPS data in a backward direction, e.g., the `rawData` and `getRawData()`, and protect them in the TEE.

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 Listing 3 to

```
1 /* Case2/drone.dtee */
2 FROM DroneAPP/main.c FUNC main {
3   TZ_DATA_CONF gpsData;
4   INSERT_AFTER [ANNT]
5     STRING_TZ *private_key;
6     TZ_genKey(RSA_KEYPAIR, 2048, private_key, NULL);
7     char *signed_gpsData;
8     TZ_sign_rsa_sha1(signed_gpsData, private_key, &
        ↪ gpsData, ...);
9   END_INSERT
10 }
```

Listing 3: D-LANG code snippet of Case 2.

achieve the core implementation of Alidrone. To insert new code logic, we use `INSERT_AFTER` and `END_INSERT` to declare an insertion block. Further, the developer can use the same annotation as in D-LANG where the original code needs to be inserted (e.g., `[ANNT]` in line 4 of Listing 3), which dTEE can handle in the static analysis. The insert statements will be placed after line 5 of the original code. The `private_key` is declared with `STRING_TZ`, specifying it is a variable in the TEE. The `signed_gpsData` (line 7 of Listing 3) 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 dTEE to facilitate developers. The former function can generate mainstream symmetric and asymmetric keys (e.g., AES and RSA) and the latter function can perform encrypt operation. Specifically, in this case, we generate a 2048-bit RSA private key and sign the GPS data in the TEE to protect the data integrity.

② **Code analysis**. In this phase, dTEE identifies sensitive statements associated with declared data and functions to meet security requirements, safeguarding the integrity or confidentiality of sensitive elements. Utilizing the application's source code, dTEE conducts static program analysis, producing a program dependency graph (PDG) that encompasses data and control dependencies. Once sensitive data is designated in the D-LANG files, dTEE treats them as tainted sources or sinks for protection. Integrating user input with the PDG, both data-flow and control-flow analyses pinpoint tainted statements destined for TEE partitioning.

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 §5.2.

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 §5.3. 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 §6.

5 DESIGN OF DTEE

5.1 Declarative Development Language

To address the limited expressiveness problem of existing TEE-development approaches [29, 42], 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 adding *new trusted logic* to complicated applications and provides *tiered protection* of variables and functions that are 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. Existing works focus on desktop applications but neglect 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 address this limitation, D-LANG incorporates the `@DRIVER` keyword, delineating the specific profile of the hardware platform. Consequently, supported non-secure device drivers for IoT applications can seamlessly metamorphose into their secure counterparts. Additionally, dTEE can auto-configure IoT devices via TZASC (e.g., TZC-400 on Hikey960 [53]), designating the peripheral as trusted and restricting REE access.

```
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 §4 shows, the `INSERT_AFTER` (line 4 of Listing 3) and `END_INSERT` (line 9 of Listing 3) are a pair of configuration statements for inserting arbitrary new program logic that is not in the original code. We showed the example in Listing 3.

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 Listing 3 shows, the `private_key` is declared with `STRING_TZ`, specifying it is a variable in the TEE. The `signed_gpsData` 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 Listing 3) function to generate a user-specified key and the `TZ_sign_rsa_sha1()` (line 8 of Listing 3) function to sign the data

```
1 FROM DroneAPP/main.c FUNC main {
2 // To permanently store the data in secure storage
3   TZ_STORE gpsData;
4 }
```

```
1 int encrypt_sub(/*...*/) {
2   // substitution implementation
3 }
4 FROM DroneApp/main.c FUNC main {
5   TZ_FUNC_SUB parseRawData ((/**/) encrypt_sub(/**/);
6 }
```

Listing 4: Illustration of logic-based program declaration of D-LANG.

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 [29, 42] 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.

Our analysis identifies that non-tiered annotation of sensitive data is insufficient for IoT development needs. For example, in the autopilot scenario, where safety is a critical topic, the central controller desires the vehicle to be controllable 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 [22] 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_CONF`. This keyword protects the declared data and other tainted data and functions that forward the data flow for confidentiality. (4) `TZ_DATA_INTG`. As shown in Listing 2, 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_CONF` and `TZ_DATA_INTG` for both confidentiality and integrity.

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. For example, as shown in Listing 4, the `parseRawData()` function is a non-TEE-based function in the original code of Alidrone (Listing 1), 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.

Table 1: D-LANG keywords overview

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

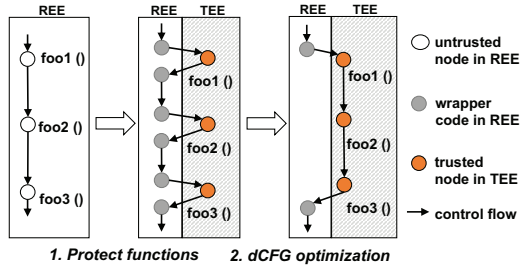

Figure 3: dCFG optimization.

5.2 dCFG-based Code Generation

After the developer uses D-LANG to declare their demands of sensitive function or data protection, dTEE needs to automatically find them and their dependencies into the TEE to protect them. A straightforward approach is to execute the entire application within the TEE [32], but this is constrained by the limited secure memory (see §7). An alternative approach is to fine-grained partition the program based on security-sensitive statements. However, our observation is that the world switching overhead is a considerable proportion.

Through the measurements on Raspberry Pi 3B+, we conclude that three times of world switch could occur the execution overhead of ~150 microseconds. As shown in Figure 3, if these five calls could be packaged into one in the TA, the world switching overhead would be reduced to ~42 microseconds. Our *key insight* is that if multiple statements such as consecutive function executions, can be bundled together and placed within the TEE without compromising security, it can reduce the overhead associated with world-switching. Thus, we propose a dCFG-based code partitioning mechanism to obtain the optimized code partition by considering the world switching overhead. This mechanism contains the following 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. In order to highlight the advantages of dCFG in analyzing code logic, we first introduce the following example.

We adapted the drone application in Listing 1 by adding more control statements to the original application, shown in Listing 5. It exhibits a mainly characteristic feature of IoT applications, namely, its primary logic is always in the `while` loop. If the drone's mode is set to low power, it will cease sampling the GPS; otherwise, it will continuously sample the GPS. Let us now consider the following

```
1  /* DroneApp/main.c */
2  int main() {
3    /* ... */ // Initialization
4    while(1) { // Primary logic
5      if (mode == "Low Power") {
6          lowPowerMode();
7          printf("Save energy...");
8          sleep(1000);
9      } else {
10         rawDataType rawData = getRawData();
11         GPSDataType gpsData = parseRawData(rawData);
12         /* ... */
13     }
14   }
15 }
```

Listing 5: The core application logic of an adapted DroneAPP.

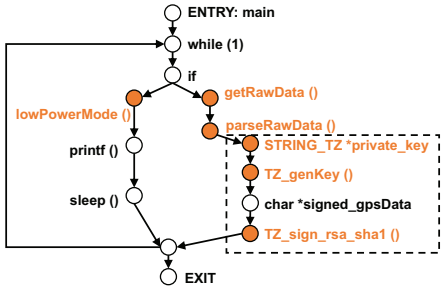

Figure 4: The dCFG construction for Listing 5. The nodes within dash block are new constructed by dTEE.

security requirements of the developers: not only do they wish to protect the GPS data collection process as in the D-LANG of Listing 3, but they also aim to safeguard the `lowPowerMode()` function to prevent attackers from disrupting the drone's transition into low-power mode.

We 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 graph 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 `INSERT`, according to the code line number and code logic of `INSERT`, insert the corresponding node in the corresponding position of the graph (e.g., the `signed_gpsData` node in the Figure 4), which indicates that CA called the TA function. The 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`, we mark the statements in the CFG diagram according to these keywords.

We generate a dCFG for this example according to the *rules*, as Figure 4 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` 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 `lowPowerMode()` node will be marked because it is a trusted function.

Partitioning algorithm. Following the described procedures, we derive a graph of statements that encapsulates all developer specifications. This graph is characterized as a directed acyclic graph $G(V, E)$, with vertices denoting logic blocks and edges signifying data flow. Our partitioning challenge is formulated as a numerical optimization, aiming to optimally allocate each logic block to either the secure or non-secure realm, adhering to security objectives. The placement of a logic block is denoted by a binary indicator X_{b_i,w_i}.

We formulate our partitioning problem as a numerical optimization problem, striving to allocate logic blocks to either the secure or non-secure world in adherence to security objectives. We use a binary indicator X_{b_i,w_i} to denote the placement of the logic block.

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

where w_i represents the possible placement world of block b_i.

We assign a sensitivity level to each sensitive variable. We define a set $S = \{s_0, s_1, ..., s_k\}$, where s_i is the ith sensitivity level.

$$Y_{b_i s_j} = \begin{cases} 1 & \text{logic block } b_i \text{ uses a variable with sensitivity } s_j \\ 0 & \text{logic block } b_i \text{ does not use variables with sensitivity } s_j \end{cases}, \quad (2)$$

Let a full path be denoted as p, spanning from a source to a sink vertex. Using $len(p)$, $\delta(p)$, and $P(G)$, we represent the path's length, its vertex count, and the complete path set in graph G, respectively. Our optimization objective is represented as $minmax_{p\in P(G)} len(p)$. Given that $len(p)$ aggregates latencies of data processing and transmission across placements, we define our goal with the binary indicator $X_{b_i,w_i} \in 0, 1$ as:

$$\begin{aligned} \arg\min_X \max_{p\in P(G)} & \sum_{i=1}^{\delta(p)} \sum_{w_i \in W_i} X_{b_i w_i} T^{Comp}_{b_i w_i} \\ & + \sum_{i=1}^{\delta(p)-1} \sum_{\substack{w_i \in W_i \\ w_{i'} \in W_{i'}}} X_{b_i w_i} X_{b_{i'} w_{i'}} T^{Trans}_{b_i w_i w_{i'}} \end{aligned} \quad (3)$$

subject to:

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

$$Y_{b_i s_j} \le X_{b_i w_{sec}}$$

where i and i' denote adjacent vertices in path p, such that $i' = i+1$. W_i indicates potential placements (secure or non-secure world) for the i-th logic block. $T^{Comp}_{b_i w_i}$ and $T^{Trans}_{b_i w_i w_{i'}}$ respectively define data processing time for block b_i at placement w_i and transmission time between blocks b_i and b'_i across placements w_i and w'_i. We posit that transmission time is trivial if consecutive blocks occupy the same world. Thus we have:

$$T^{Trans}_{b_i w_i w_{i'}} = \begin{cases} r_{ii'} t_{ii'k} & w_i \neq w_{i'} \\ 0 & w_i = w_{i'} \end{cases}, \quad (4)$$

where $r_{ii'}$ denotes the data size transmitted across edge (i, i'), and $t_{ii'k}$ is a method-specific factor quantifying the transfer time using method k between blocks i and i'. Data can be transmitted between worlds through two predominant methods: register-based and shared-memory-based transmissions. 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 after optimization.

The optimization problem introduces further *constraints*: *(i)* given the restricted secure memory in TrustZone, the aggregate memory use, quantified as $\sum_{i=1}^{\delta(p)} \sum_{w_i \in W_i} X_{b_i w_i} m_{b_i}$ for all logic blocks in the secure domain, cannot exceed the secure world's capacity M. m_{b_i} denotes the maximum memory allocation for a given logic block. *(ii)* dTEE must respect user specifications to place particular blocks into the secure world due to tiered variable considerations.

However, the problem formulation presented in Equation (3) characterizes a quadratic minimax dilemma, proven to be NP-hard in complexity [14]. Thanks to the McCormick relaxation, we can re-formulate the problem to make it follow standard integer programming problem (ILP) formulation. We introduce auxiliary variable $v = X_{b_i w_i} X_{b_{i'} w_{i'}}$ and z, to convert the inner max function to a set of constraints to make it follow standard ILP formulation. The re-formulated Equation (3) is illustrated as:

Objective: $\arg\min_X \quad z$

Subject to:

$$z \ge \sum_{i=1}^{\delta(p)} \sum_{w_i \in W_i} X_{b_i w_i} T^{Comp}_{b_i w_i} + \sum_{i=1}^{\delta(p)-1} \sum_{\substack{w_i \in W_i \\ w_{i'} \in W_{i'}}} v_{i s_i s_{i'}} T^{Trans}_{b_i w_i w_{i'}}, \forall p \in G. \quad (5)$$

The constraints of v are linear [14], and thus we can efficiently solve the function by the standard solver, e.g., `lp_solve`.

It is worth noting that the dCFG optimization process is capable of targeting multi-TAs. dCFG can set the safe memory for different applications and solve it by considering the constraints of multiple applications. Since existing mainstream TEE OSes (e.g., OP-TEE, TinyTEE, etc.) for mobile and IoT devices do not support multi-thread TA, it is not feasible to directly use multi-thread libraries such as `pthreads`. We suggest one way to execute multi-thread TA is instead by running two TAs concurrently.

5.3 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

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` [11], `Libnmea` [24]	`LibwiringPi` [19]	`Libopenssl` [35], `Libcrypto` [35]
Konstantin et al. [42]	✓	✘	✘
Glamdring [29]	✓	✘	✘
dTEE (Our paper)	✓	✓	✓

physical interaction for functionalities, such as r/w GPIOs and sampling sensors. Since existing works lack concern about peripheral interactions, there is no solution adaptable to IoT applications today. Furthermore, we systematically 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.

Portable libraries. This type of library is the simplest type of protection. These library files are restricted to all the functions supported by the TEE OS and no Linux system calls (e.g., `open()`, `close()`, `write()`) invocation. The prevalent TEE operating systems typically support a subset of the Standard C Library. Consequently, portable libraries within these environments are subject to stringent constraints.

For example, the Quirc [11] 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 code of the 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 [29, 32, 42].

Auto-transformable libraries. This type of library is more complex than portable libraries because of its interaction with the physical world using peripherals. IoT devices can be dichotomized into: *(i)* MMU-equipped devices employing memory-mapped techniques to access hardware resources, e.g., GPIO and registers, facilitated by mapping physical to virtual addresses, often via the `mmap` interface; and *(ii)* non-MMU devices directly interfacing with hardware through physical address manipulation. dTEE is applicable to both aforementioned device categories. For the latter, the transformation of peripheral libraries is straightforward, given that the manner in which the peripheral library controls physical hardware addresses in the non-secure world mirrors its operations in the secure world, thus inherently supporting automatic transformation. However, for the former, the process is more intricate due to the absence of interfaces akin to `mmap` in TEE OSes.

For example, LibwiringPi [19] is a prominent C/C++ peripheral library for the renowned IoT platform Raspberry Pi, principally relies on the `mmap` function for GPIO operations (e.g., `digitalWrite()` and `digitalRead()`). Directly porting these functions into the TEE is unfeasible because a TA operating within the TEE's user space lacks permissions for physical address mapping to secure memory, a privilege reserved for the TEE kernel. As a solution, dTEE extends the basic mapping mechanism for TEE OSes (e.g., the `phys_to_virt()` function in OP-TEE) and offers pre-configured kernel-level functions for GPIO access. Specifically, dTEE incorporates foundational mechanisms within the TEE OS kernel to translate physical addresses to virtual ones for peripherals. Upon receiving the base hardware addresses from developers, dTEE proficiently transforms non-secure peripheral libraries into *shadow* trusted libraries utilizing the pre-configured functions. While the interface may be exploited by malicious applications to access trusted peripherals, dTEE can enhance security by integrating access control measures to obstruct such unauthorized calls. It is highlighted that dTEE's primary objective is to alleviate the burden on developers by securing existing IoT applications, exploring inspection mechanisms is out of our scope.

Manually-transformable libraries. This type of library is the most complex situation due to the program logic has changed. In situations where developers seek to introduce secure program logic to established IoT applications, D-lang serves as a pivotal tool for incorporating these security augmentations. Directly porting applications that rely on Linux file system calls or substantial TCB size-increasing libraries, such as the OpenSSL library [35], to a TEE proves impractical without requisite alterations. However, dTEE addresses these complexities through the utilization of replacement and insertion mechanisms inherent to D-lang.

We elucidate this with two real-world examples and demonstrate how dTEE offers countermeasures. The first example is MQTTZ [43], which employs software encryption from OpenSSL to encrypt client messages. If developers intend to switch to hardware chip-accelerated encryption, merely safeguarding the original non-secure encryption within the TEE becomes redundant due to the altered encryption logic. Another example, DarkneTZ [33] modifies the DNN inference logic of a pre-existing framework [40]. It introduces novel program logic, designating partition points to distinguish between sensitive layers executed in the TEE and standard layers in the REE. In such contexts, adhering to the logic of original codes, as seen in [29] and [42], becomes moot due to the incorporation of new security logic. To tackle the aforementioned problems, the developers can use dTEE with the following solutions respectively: *(i)* users could specify the library of cryptography to

use, such as `Libmbedtls` and `LibTomCrypt`, or hardware accelerating chips, *(ii)* users could use insert statements that add their new trusted logic. We will show details in §7.1.

6 SYSTEM IMPLEMENTATION

In this section, we will describe the implementation details and the portability of dTEE. The code analysis of dTEE is built as an extension to the Frama-C [1], 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 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 [2, 3] 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 [16], 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, lacking the capability to sustain basic REE and TEE sessions. To bridge this gap, dTEE synthesizes auxiliary code. To ensuring TEE independence, we adopt standard TEE Client and Internal APIs as standardized by GlobalPlatform [18] and adopted by prevalent TEE OSes like OP-TEE [28] and QSEE [39]. Typically, the CA and TA consist of five skeleton functions to maintain a connection between CA and TA. We produce paired initialization, session, and finalization codes for the CA and TA. Subsequent command invocations are tailored based on CA-to-TA function calls. Given the stringent datatype definitions in the TEE-based programming model, dTEE scrutinizes the security of data flows, leaning on constant data types. Moreover, we generate unique identifiers for each TA situated within the TEE.

On the non-secure world side of the application, auxiliary code is generated from caller parameters and returns. Given the four-parameter limit between CA and TA, dTEE utilizes native APIs, like `TEEC_VALUE_INPUT` and `TEEC_MEMREF_TEMP_INPUT`, for functions with four or fewer integer or string parameters. For more extensive requirements, dTEE reserves shared memory for parameter serialization, with subsequent deserialization in TA. However, pointers in C applications present vulnerabilities. While it might seem practical to dereference pointers passed from CA and re-reference them in TA, pointers can be weaponized for attacks, including confused-deputy and TOCTOU (Time of Check, Time of Use) attacks [25]. Hence, we undertook rigorous pointer sanitation. To counteract confused-deputy attacks, we encapsulate dereferenced pointers in the TEE with objects controlling their lifespan and access, preventing direct memory access. To preserve function logic while resolving pointer-related concerns, consider a parameter passed from CA to TA—a pointer to a structure with an integer and string. We initially dereference this structure pointer and allocate a designated block of shared memory. Subsequently, we serialize the integer and string into this memory. This serialized data is then conveyed to the TEE. Within the TA, we reconstruct an analogous structure to mirror the CA's variable structure, enabling us to deserialize the parameters without altering the TA's existing codebase. Consequently, the TA's code executes seamlessly with the re-referenced, deserialized parameters. We implement the serialization and deserialization mechanism in the code generation stage. Specifically, it is implemented in CA and user TA to avoid injection attacks by malicious kernels. dTEE also conducts security checks to ensure the integrity before transmitting it to the TA.

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. Specifically, dTEE offers standardized APIs across diverse hardware platforms, ensuring that both inherent peripheral functions and sensitive drivers are deemed trustworthy. These intrinsic functions remain agnostic to the TEE OS. For example, on the Raspberry Pi 3B+ platform, the function `TZ_digitalWrite()` endeavors to configure the pin to the "output" mode, referencing the addresses of GPFSEL, GPSET, and GPCLAR registers as per the documentation [12]. While the logic of built-in functions is contingent upon the hardware and independent of the TEE, their generated code is TEE-specific. As an illustration, in the TEE known as OP-TEE, the inherent peripheral code is instantiated in `optee_os/core/arch/arm/pta` for pseudo-TAs [48].

7 EVALUATION

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

In our experiments, we explore the capabilities of an open source TEE, OP-TEE [28], implemented on a widely used IoT platform, Raspberry Pi 3B+ board [37]. We installed Raspbian Kernel Version 4.14.98 as the REE OS and OP-TEE Version 3.4.0 as the TEE OS. We selected this board because the platform has been widely evaluated for assessing TEE applications in prior studies. Despite the absence of several optional TrustZone components (e.g., TZASC) in RPi 3B+, the influence on our evaluation results is negligible for two reasons: (1) dTEE is primarily concerned with the partitioning of existing applications, thus the absence of these components does not affect its functionality; (2) a recent work [21] has enabled the implementation of the security component's functionality at the software level on the RPi 3B, and resulting in a negligible overhead (about 1% for some applications).

7.1 Case Study

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 §3. For MQT-TZ [43], TZ4Fabric [34], and DarkneTZ [33], we implement their core functionality while leaving out less relevant details. For Alidrone [30], due to the lack of source code, we implement it by the main idea and techniques in the literature. We

Table 3: 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. [42]	Glamdring [29]	WATZ [32]	**dTEE**
Print	5	3	3	2	7.8K	12.6K	98K	✓	✓	✓	✓
cJSON [17]	19	1	1	2	31K	12.4K	112.4K	✘	✓	✓	✓
Concat	33	3	3	2	7.9K	6.1K	100K	✓	✓	✓	✓
Blink [19]	21	1	1	2	8.3K	12.7K	97.7K	✘	✘	✘	✓
Temp [19]	181	4	4	4	8.5K	12.7K	98K	✘	✘	✘	✓
Humi [19]	181	4	4	4	8.5K	12.7K	98K	✘	✘	✘	✓
Alidrone [30]	129	11	9	2	29K	12.6K	121.4K	✘	✘	✘	✓
MQT-TZ [43]	291	5	5	2	13K	12.6K	100K	✘	✘	✘	✓
TZ4Fabric [34]	89	8	6	2	8.5K	12.7K	108.2K	✘	✓	✓	✓
DarkneTZ [33]	33.5K	31	31	4	366K	384K	112K	✘	✘	✘	✓
Socket [27]	118	N/A	N/A	N/A	34K	N/A	N/A	✘	✘	✘	✘

```
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) {
  return TZ_enc_aes_cbc(plain_text, plain_text_len,
    ↪ key_id, iv_id, cipher_text, key_size);
}
FROM mqt-tz/hot_cache/main.c FUNC main {
  TZ_STORE dest->iv, dest->key;
  TZ_FUNC_SUB encrypt(/**/) encrypt_sub(/**/);
}
```

Listing 6: Expressing MQT-TZ with D-LANG.

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 instantiated MQT-TZ using `libopenssl`, integrating `save_key` and `aes` within MQT-TZ brokers for benchmarking. The broker retains each client's key and encrypts data for client transfer. The `mqttz_client` struct comprises four elements: `cli_id` (client ID), `iv` (initial vector for symmetric key), `key` (symmetric key), and `data` (payload). Data encryption employs a 128-bit AES key. As illustrated in Listing 6, we fortified MQT-TZ's application by utilizing six keywords: two for sensitive data localization, two for `iv` and `key` storage, and two for replacing the encryption with `TZ_enc_aes_cbc()`.

In MQT-TZ, the AES `key` and `iv` values for the client necessitate persistent storage within the TEE's secure domain. Consequently, we employ `TZ_STORE` statements for these variables to facilitate this storage. Notably, the original non-secure version utilizes `encrypt()` from `libopenssl`. To ensure encryption security, MQT-TZ replicates the `encrypt()` function within the TEE. With D-LANG, this adaptation requires merely three lines of code, attributed to TEE's cryptographic patterns being identified as dTEE templates, which furnish built-in functions.

TZ4Fabric. TZ4Fabric is engineered to run smart contracts within TEEs. In our rendition, we spotlight three smart contract functions: `create()`, `add()`, and `query()`. Using D-LANG and the `TZ_FUNC` keyword, as illustrated in Listing 7, we emulate TZ4Fabric's security framework. This keyword extends security to *subfunctions* as well; within `create()`, subfunctions like `create_get_state()`, `create_put_state()`, and `create_write_response()` are invoked. Leveraging call graphs, dTEE discerns and isolates pertinent statements to the secure domain, ensuring the integrity of the smart contract operations within the TEE.

```
FROM chaincode_proxy/main.c FUNC fabric {
  TZ_FUNC create(), add(), query();
}
```

Listing 7: Expressing TZ4Fabric with D-LANG.

```
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 [A1]
  /* ... */
  END_INSERT
}
FROM darknet/src/parser.c FUNC load_weights_upto {
  INSERT_AFTER [A2]
  /* ... */
  END_INSERT
}
```

Listing 8: Expressing DarkneTZ with D-LANG.

DarkneTZ. DarkneTZ aims to protect sensitive DNN model layers within the TEE against privacy breaches. Our focus is on DNN inferencing. The DarkneTZ implementation incorporates 31 D-LANG keywords, detailed in Listing 8. Specifically, 20 `TZ_FUNC` keywords delineate sensitive data, such as `make_softmax_layer`, while pairs of `INSERT_AFTER` and `END_INSERT` embed trust logic within `network_predict()` and `load_weights_upto()`. Contrasting the original, the enhanced security strategy executes non-sensitive DNN layers in the non-secure domain and the sensitive layers securely. This is achieved by leveraging `INSERT_AFTER` and `END_INSERT` within the weight-loading (i.e., `load_weights_upto()`) and prediction (i.e., `network_predict()`) phases to integrate new logic. This involves loading network weights from both REE and TEE and isolating the final five layers within the TEE. Compared to prior methods, our refined approach to integrating programming logic has proven especially effective in this case study.

Alidrone. We introduced two usage examples in §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` [24]. 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

```
@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, Privatekey2048_D,
      ↪ Privatekey2048_N;
  TZ_DATA_CONF data;
  TZ_FUNC_SUB EvpSign() EvpSign_sub();
}
```

Listing 9: Expressing Alidrone with D-LANG.

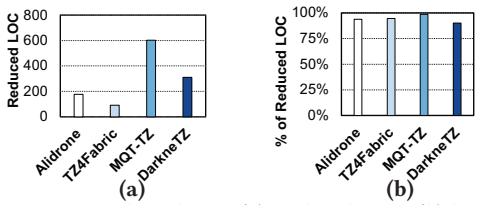

Figure 5: Reduced LOC (a) and code rate (b) by dTEE.

TEE. As Listing 9 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_CONF` to protect the subsequent operations.

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 LibwiringPi [19] that contains LED blinks, reading temperature, and humidity; on the other hand, we use a small library cJSON [17] 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 [27] for sending messages by networks.

Results of benchmarks. Glamdring [29] is tailored for SGX, whereas [42] is designed for Android applications. While a direct comparison of execution time with dTEE may not be equitable, their expressiveness can be juxtaposed. Table 3 shows the results of benchmarks and Figure 5 shows the reduced LOC of real-world applications. The checkmark entailed in Table 3 means feasibility and guarantees for security. Specifically, observations are as follows. *(i)* On the expressiveness, as Table 3 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 §5.3), dTEE can provide automatic transformation of the prevalent sensor drivers (e.g., IMU and temperature & humidity sensors) of various IoT devices to the trusted ones. Nonetheless, if the partitioning of an existing application notably expands the Trusted Computing Base (TCB) attack surface, dTEE abstains from conversion and warns developers. More complex peripherals (e.g., NIC), which necessitate sophisticated library drivers, cannot be directly transformed by dTEE, because these peripherals necessitate sophisticated library drivers thereby requiring large TCB (e.g., the Socket application in Table 3) [26, 51]. Fortunately, these complex peripherals can be implemented through the record-replay mechanism [20], i.e., recording the interaction between the CPU and MMIO in the peripheral driver offline and replaying them in the TEE online. While the current prototype of dTEE lacks implementation of this mechanism, it is designed to be complementary to the existing automatic library conversion mechanism.

Table 4: Comparison with the existing work w.r.t execution efficiency. The results are evaluated and averaged on the `speedtest1` benchmarks of SQLite [45].

	Execution time		Memory usage	
	Overall	Switching	Overall	Wasm runtime
WATZ [32]	2,160 ms	/	12.10 MB	9.15 MB
dTEE (w/o dCFG opt)	1,217 ms	460.27 ms	2.14 MB	/
dTEE (w/ dCFG opt)	1,050 ms	293.03 ms	2.89 MB	/

(ii) On the lines of code reduction, as shown in Figure 5, we find that dTEE can reduce more than 90% lines of code in real-world IoT applications compared to developers using OP-TEE alone. Since D-LANG has a lot of built-in functions of peripherals and cryptography, dTEE achieves few keywords and lines of code of D-LANG to represent the security demands of trusted IoT applications, shown in Table 3.

7.3 dTEE vs. WATZ

In order to demonstrate the advantages of utilizing dTEE, a comparative analysis with WATZ [32] is presented. WATZ is the state-of-the-art work in automatically transitioning existing applications to become compatible with TrustZone. Similarly, both dTEE and WaTZ are focused on securing applications into TEE-enabled ones that comparison between them is suitable. Note that WATZ compiles the entire application code into WebAssembly bytecode and executes it in the TEE without modifying the code. To enable WebAssembly execution in the TEE, WATZ ports the WAMR runtime [4] into the TEE, significantly increasing the TCB size. Evaluating on QEMU v8, as indicated in Table 4, reveals that WATZ's WAMR runtime occupies ~9MB of secure memory (~69% of QEMU v8's total secure memory) even without executing any application. It is notable that secure memory is a scarce resource, even on high-end IoT devices, such as the Hikey960 development board, which has a maximum available secure memory of 64MB.

We next compare dTEE and WaTZ in the context of a mobile widespread database engine (SQLite [45]), focusing on secure memory usage, execution time, and code modifications. Our analysis focuses on the `speedtest1` [46], a performance benchmark for SQLite that encompasses diverse input datasets and SQL statements. As illustrated in Table 4, we averaged the results for each benchmark, contrasting WaTZ and dTEE both pre- and post-implementation of dCFG-based optimization (cf. §5.2). In terms of execution time, we observed that: *(i)* Before performing dCFG-based optimization, dTEE reduces at 44% of WATZ due to its native execution, whereas WATZ executes WebAssembly bytecode, inherently incurring a

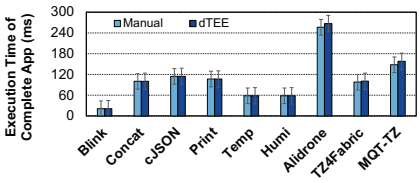

Figure 6: Complete execution time of TEE-based applications. The development approaches of manual and dTEE are compared.

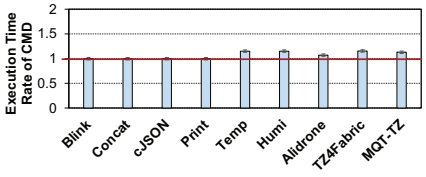

Figure 7: CMD execution overhead rate between original application and dTEE applications. The red line (—) represents the manual approach.

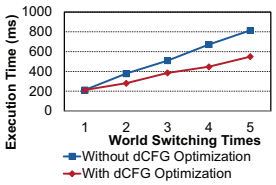

Figure 8: Execution time of different world switching times with/without dCFG.

1.7~2x computational overhead, aligning with the experimental results of WATZ and measurement work [50]. *(ii)* After performing dCFG-based optimization, dTEE reduces execution time by 13.7% compared to it without optimization, attributable to the diminished time spent switching between REE and TEE. In terms of memory usage, we observed that: *(i)* WATZ's WAMR runtime occupies ~9MB of secure memory even without executing any application. *(ii)* After reducing the runtime memory of WAMR, the WATZ memory usage is comparable to that of dTEE. Besides, owing to the dCFG-based optimization, which aims to achieve optimized execution time, more continuous database operations are placed in the TEE, thus attaining ~1.16x the secure memory usage prior to optimization.

For the `speedtest1` benchmark, dTEE only requires the addition of 33 lines of DLANG code to protect each benchmark's sensitive variables and functions, while the entire `speedtest1` and SQLite comprises 25.2K lines of code. Consequently, we can conclude that dTEE can achieve faster execution times and smaller secure memory usage than WATZ by modifying a minimal number of code lines.

7.4 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.

Table 3 shows the binary size of CA and TA. While CA exhibits minimal memory overhead relative to the original applications, TA displays an enlarged binary size due to its incorporation of numerous TEE internal functions and linkage with static libraries. It is noteworthy that only D-LANG's built-in functions yield slight memory increases compared to the manual approach. This is because the manual method allows developer customization within applications, such as the integration of specialized libraries, a capability absent in dTEE's inherent functionality. Nonetheless, the adaptation and enhancement of dTEE's built-in functions can be achieved through continuous revisions of its implementation version.

In Figure 6, we depict the comprehensive execution times of benchmarks, while Figure 7 focuses on the core function command (CMD) execution times, e.g., `TEEC_InvokeCommand()`. The manual method in the experiment in Figure 6 is for developers to use OP-TEE directly. The performance of dTEE aligns closely with the manual approach, incurring a mere 6.6% execution overhead at its peak. For micro-benchmarks, both methods are almost indistinguishable in execution time, primarily because dTEE-generated glue code mirrors that of the manual counterpart. However, in intricate real-world applications, dTEE's execution time lags slightly due to its universal, and potentially excessive, auxiliary code for bridging REE and TEE. An instance is MQT-TZ, where a manual technique can cache keys rather than re-triggering a read API with every encryption, as done by dTEE. This limitation, rooted in dTEE's generic design, can potentially be mitigated by auto-detecting and eliminating superfluous code.

7.5 Performance Optimization

We demonstrate the improved performance of dCFG-based code partition (§5.2) on the Alidrone [30]. Specifically, 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 8 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 1.35x~1.48x. 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.

7.6 Discussion

Support for other TEEs. dTEE is designed for ARM's TrustZone to reflect its dominance in mobile and IoT markets. However, dTEE presents feasibility for adaptation to support other TEEs, e.g., Intel's SGX, a TEE architecture prevalent in cloud computing and data centers. The adaptation process necessitates no changes in the code declaration and analysis phases but requires modifications in the code partitioning and generation stages due to distinct TEE APIs, notably `ecall` and `ocall`, used in SGX environments. The dCFG optimization process is applicable across various TEE platforms due to its design as a generalized optimizer. Furthermore, there is potential for dTEE to be extended to support emerging RISC-V based TEEs, such as VirtualZone [41].

Trade-off between performance and expressiveness. To support declarative development, dTEE tradeoffs performance for expressiveness to some extent. For example, the current implementation of dTEE does not support fine-grained concurrency control over the IoT device. This trade-off reflects the principle that increased expressiveness typically leads to reduced performance efficiency. Future developments in dTEE aim to introduce hierarchical tools that cater to varying developer expertise levels.

8 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.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers and the shepherd for their valuable comments and helpful suggestions. This work is supported by the National Natural Science Foundation of China under Grant No. 62072396 and 62272407, the "Pioneer" and "Leading Goose" R&D Program of Zhejiang under grant No. 2023C01033, and the National Youth Talent Support Program. Wei Dong is the corresponding author.

REFERENCES

[1] 2022. Frama-C - Framework for Modular Analysis of C programs. https://frama-c.com.
[2] 2022. Impact analysis plug-in. https://frama-c.com/fc-plugins/impact.html.
[3] 2022. Scope & Data-flow browsing plug-in. https://frama-c.com/fc-plugins/scope.html.
[4] 2023. WebAssembly micro runtime. https://github.com/bytecodealliance/wasm-micro-runtime.
[5] 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* 21, 3 (2021), 1–23.
[6] 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 (JSAC)* 39, 2 (2020), 361–369.
[7] Android. 2022. Trusty TEE. https://source.android.com/security/trusty.
[8] ARM. 2022. TrustZone for Cortex-A. https://www.arm.com/technologies/trustzone-for-cortex-a.
[9] Arm. 2023. TrustZone Address Space Controller. https://developer.arm.com/documentation/ddi0431/c/introduction/about-the-tzasc/features-of-the-tzasc.
[10] 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.
[11] Daniel Beer. 2022. Quirc. https://github.com/dlbeer/quirc.
[12] BROADCOM. 2022. BCM2835 ARM Peripherals. https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf.
[13] Ivan De Oliveira Nunes, Xuhua Ding, and Gene Tsudik. 2021. On the root of trust identification problem. In *Proc. of ACM/IEEE IPSN*.
[14] Raphael Eidenbenz and Thomas Locher. 2016. Task allocation for distributed stream processing. In *Proc. of IEEE INFOCOM*.
[15] The Eclipse Foundation. 2023. Eclipse Mosquitto - An open source MQTT broker. https://mosquitto.org/.
[16] FRAMA-C. 2022. Slicing plug-in. https://frama-c.com/fc-plugins/slicing.html.
[17] Dave Gamble. 2022. Ultralightweight JSON parser in ANSI C. https://github.com/DaveGamble/cJSON.
[18] GlobalPlatform. 2022. GlobalPlatform. https://globalplatform.org.
[19] Gordon. 2022. WiringPi. https://github.com/WiringPi/WiringPi.
[20] Liwei Guo and Felix Xiaozhu Lin. 2022. Minimum viable device drivers for ARM TrustZone. In *Proc. of ACM EuroSys*.
[21] Seung-Kyun Han and Jinsoo Jang. 2023. MyTEE: Own the Trusted Execution Environment on Embedded Devices.. In *Proc. of NDSS*.
[22] 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.
[23] Intel. 2022. Intel®SoftwareGuardExtensions (Intel®SGX). https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html.
[24] Jacketizer. 2022. libnmea: Lightweight C library for parsing NMEA 0183 sentences. https://github.com/jacketizer/libnmea.
[25] Xeno Kovah, Corey Kallenberg, Chris Weathers, Amy Herzog, Matthew Albin, and John Butterworth. 2012. New results for timing-based attestation. In *Proc. of IEEE S&P*.
[26] 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.
[27] Browny Lin. 2022. Simple socket example. https://gist.github.com/browny/5211329.
[28] Linaro. 2022. Open Portable Trusted Execution Environment. https://www.op-tee.org.
[29] 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 *Proc. of USENIX ATC*.
[30] Tianyuan Liu, Avesta Hojjati, Adam Bates, and Klara Nahrstedt. 2018. Alidrone: Enabling Trustworthy Proof-of-Alibi for Commercial Drone Compliance. In *Proc. of IEEE ICDCS*.
[31] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2021. Twine: An embedded trusted runtime for webassembly. In *Proc. of IEEE ICDE*.
[32] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2022. WaTZ: A Trusted WebAssembly Runtime Environment with Remote Attestation for TrustZone. In *Proc. of IEEE ICDCS*.
[33] 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 *Proc. of ACM MobiSys*.
[34] Christina Müller, Marcus Brandenburger, Christian Cachin, Pascal Felber, Christian Göttel, and Valerio Schiavoni. 2020. TZ4Fabric: Executing Smart Contracts with ARM TrustZone. In *Proc. of IEEE SRDS*.
[35] OpenSSL. 2022. OpenSSL. https://www.openssl.org.
[36] Nidhi Pathak, Anandarup Mukherjee, and Sudip Misra. 2020. Reconfigure and reuse: Interoperable wearables for healthcare IoT. In *Proc. of IEEE INFOCOM*.
[37] Raspberry Pi. 2022. Raspberry Pi 3 Model B+. https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus.
[38] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–36.
[39] Qualcomm. 2022. Mobile Security Solutions. https://www.qualcomm.com/products/features/mobile-security-solutions.
[40] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.
[41] RISC-V. 2023. XuanTie VirtualZone: RISC-V-based Security Extensions. https://riscv.org/blog/2022/04/xuantie-virtualzone-risc-v-based-security-extensions-xuan-jian-alibaba/.
[42] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. 2016. Automated Partitioning of Android Applications for Trusted Execution Environments. In *Proc. of IEEE/ACM ICSE*.
[43] Carlos Segarra, Ricard Delgado-Gonzalo, and Valerio Schiavoni. 2020. MQT-TZ: Hardening IoT Brokers Using ARM TrustZone. In *Proc. of IEEE SRDS*.
[44] 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 *Proc. of ACM ASPLOS*.
[45] SQLite. 2023. SQLite. https://www.sqlite.org/index.html.
[46] SQLite. 2023. SQLite Speedtest Benchmark. https://sqlite.org/src/file/test/speedtest1.c.
[47] Mingyue Tang, Guimin Dong, Jamie Zoellner, Brendan Bowman, Emaad. Abel-Rahman, and Mehdi Boukhechba. 2022. Using ubiquitous mobile sensing and temporal sensor-relation graph neural network to predict fluid intake of end stage kidney patients. In *Proc. of ACM/IEEE IPSN*.
[48] TrustedFirmware.org. 2023. Trusted Applications. https://optee.readthedocs.io/en/latest/architecture/trusted_applications.html.
[49] 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 *Proc. of ACM CCS*.
[50] Yutian Yan, Tengfei Tu, Lijian Zhao, Yuchen Zhou, and Weihang Wang. 2021. Understanding the performance of webassembly applications. In *Proc. of ACM IMC*.
[51] Kailiang Ying, Amit Ahlawat, Bilal Alsharifi, Yuexin Jiang, Priyank Thavai, and Wenliang Du. 2018. Truz-droid: Integrating trustzone with mobile operating system. In *Proc. of ACM MobiSys*.
[52] 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.
[53] Yiming Zhang, Yuxin Hu, Zhenyu Ning, Fengwei Zhang, Xiapu Luo, Haoyang Huang, Shoumeng Yan, and Zhengyu He. 2023. SHELTER: Extending Arm CCA with Isolation in User Space. In *Proc. of USENIX Security*.