Hindawi
Security and Communication Networks
Volume 2022, Article ID 8033799, 21 pages
https://doi.org/10.1155/2022/8033799

Research Article

TEE-Watchdog: Mitigating Unauthorized Activities within Trusted Execution Environments in ARM-Based Low-Power IoT Devices

Anum Khurshid, Sileshi Demesie Yalew, Mudassar Aslam, and Shahid Raza

RISE Research Institutes of Sweden, Stockholm, Sweden

Correspondence should be addressed to Anum Khurshid; anum.khurshid@ri.se

Received 5 November 2021; Accepted 7 March 2022; Published 25 May 2022

Academic Editor: Leonardo Mostarda

Trusted execution environments (TEEs) are on the rise in devices all around us ranging from large-scale cloud-based solutions to resource-constrained embedded devices. With the introduction of ARM TrustZone-M, hardware-assisted trusted execution is now supported in IoT nodes. TrustZone-M provides isolated execution of security-critical operations and sensitive data-generating peripherals. However, TrustZone-M, like all other TEEs, does not provide a mechanism to monitor operations in the trusted areas of the device and software in the secure areas of an IoT device has access to the entire secure and nonsecure software stack. This is crucial due to the diversity of device manufacturers and component suppliers in the market, which manifests trust issues, especially when third-party peripherals are incorporated into a TEE. Compromised TEEs can be misused for industrial espionage, data exfiltration through system backdoors, and illegal data sharing. It is of utmost importance here that system peripheral behaviour in terms of resource access is in accordance with their intended usage that is specified during integration. We propose TEE-Watchdog, a lightweight framework that establishes MPU protections for secure system peripherals in TrustZone-enabled low-end IoT devices. TEE-Watchdog ensures blocking unauthorized peripheral accesses and logging of application misbehaviour running in the TEE based on a manifest file. We define lightweight specifications and structure for the application manifest file enlisting permissions for critical system peripherals using concise binary object representation (CBOR). We implement and evaluate TEE-Watchdog using a Musca-A2 test chipboard. Our microbenchmark evaluations on CPU time and RAM usage demonstrated the practicality of TEE-Watchdog. Securing the system peripherals using TEE-Watchdog protections induced a 1.4% overhead on the latency of peripheral accesses, which was 61 microseconds on our test board. Our optimized CBOR-encoded manifest file template also showed a decrease in manifest file size by 40% as compared to the standard file formats, e.g., JSON.

1. Introduction

The Internet of things (IoT) is a network of data-generating and data-consuming devices encompassing critical infrastructures of future systems. The data generated by these devices are typically sensitive in nature as it can be personal data generated from personalized healthcare systems (health monitoring devices), confidential data from industrial control systems revealing information from the operations, or privacy-critical user data from home monitoring systems. We observe a fast-paced development of the application frameworks for these systems such as Samsung SmartThings [1] and Apple HomeKit [2]. However, most of the applications themselves are not designed with security in mind but remain focused on performance and functionality raising several security concerns. The IoT devices build upon and enable third-party component suppliers (such as sensors, cameras, and microphones) and application developers to be a part of the underlying framework. The heterogeneity of these component suppliers and the lack of security standards and privacy-preserving protocols hinder the trusted computing base (TCB) of IoT devices. When such a third-party component/peripheral is integrated into the existing framework, it introduces potential risk due to the accompanying software stack/driver [3]. Hence, the data collected by these peripherals are also a concern for device owners due to privacy reasons. Ditio [4] proposes a solution to address this problem for smartphones and gives security assurance to users. It records system peripheral activity (like sensors), which can later be inspected by an auditor for compliance with user-defined policies. Trusted execution environments (TEEs) such as ARM TrustZone [5], Intel SGX [6], and Keystone [7] provide a mechanism to partition system resources and peripherals into secure and nonsecure processing environments to enable isolation of critical components from the rest of the system. Hence, using TrustZone we can separate security-critical operations such as key management, crypto-operation, DRM, sensitive peripherals, and their associated drivers into the secure areas of device. The system components in the secure areas also have access to the entire system resources. This hardware-based isolation provides strong security guarantees to device users and owners.

2. Challenge

If a secure application is compromised (due to an undetected vulnerability), it becomes very challenging to detect this compromise and deploy remedial actions. Several examples of implementing stealthy rootkits in such compromised TEEs [8, 9] exist. Thus, the device users trust that software in secure areas of the device is not buggy, has backdoors, or will not abuse the privileged access to manipulate or exfiltrate data since they can also have access to the network stack. The fundamental problem is that in a multi-vendor environment, the device manufacturer has to ensure *vendor trust* (i.e., component vendors trusting each other for data handling) and *user trust* (i.e., trust of user on the device manufacturer even though individual components of the device have distinct origins). Industrial espionage in case of conflict of interest between vendors and illegal file sharing using compromised secure areas are some potential ways TEE-enabled systems can be exploited. The need for a solution that forces peripherals and their software to behave as agreed upon during integration cannot be more emphasized. In case of conventional IoT devices including smartphones, smart TVs, and smart vehicles, the component vendors and suppliers are fewer and well-known. Moreover, the security of the underlying architecture and applications is matured over years of research and development. On the other hand, IoT device vendors are still emerging rapidly, resulting in the spread of unregulated IoT devices. With the introduction of TrustZone-M (TZ-M) for Cortex-M-based devices (i.e., low-end IoT nodes), the hardware-based TEE for isolated execution of critical operations is available to the IoT world as well. This enables the separation of critical components from system operations, but the problem discussed above is now equally applicable to low-end devices. As a result, introducing these devices in our proximity without well-defined trust anchors threatens our privacy and safety. The major problem posed here is that software running in the secure areas of a TrustZone-M enabled device lacks fine-grained access to the system peripherals. This raises concerns about the intended usage of the installed peripherals and the way they are actually used. Running conventional antivirus systems increases the code base of secure areas, which is impractical, and hence, these methods are not suggested.

In this study, we present TEE-Watchdog (shown in Figure 1), a framework to map user/vendor-defined policies; i.e., CBOR policy associated with peripherals (fingerprint sensor in Figure 1) to the system memory, efficiently detect access violations, and register policy-violating application's behaviour, which after audit helps select appropriate remedial actions. We implement a *Security Manager* comprising of a *policy converter*, a *policy enforcement module,* and a *logging module*. We introduce an optimal structure for access policies in the *manifest file*, an *access table,* and a system *log file* containing the necessary information for analysis of malicious behaviour. TEE-Watchdog's optimized design is implemented with minimal modifications to the secure fault handlers and bootloader. TEE-Watchdog enforces memory accesses based on policies using a memory protection unit (MPU). TEE-Watchdog adds system structures such as *manifest file*, *access table,* and *log file* to establish the entire framework. With the combination of these mechanisms, besides providing a runtime secure peripheral access monitoring and behaviour logging mechanism for the secure peripherals in TrustZone-M-enabled IoT, we ensure the following: (i) fine-grained access control over secure system peripherals, (ii) immutability of proposed system components, i.e., the *Security Manager* and *access table*, and (iii) confidentiality and integrity of the *log file*.

3. Contributions

The main contributions of our TEE-Watchdog paper are as follows:

(i) A mechanism to protect against unauthorized, curious access of critical resources by trusted applications in the TrustZone-based TEE of IoT devices.

(ii) We propose a lightweight manifest file specification (in CBOR), for trusted apps running in TEE.

(iii) We propose an automated translation mechanism to convert application's access policies into a platform-dependent and tamper-resistant access table.

(iv) Exploiting the MPU, we provide detection, blocking, and behavioral logging of the policy-infringing suspicious application within TEE.

(v) We implement and evaluate TEE-Watchdog using real IoT hardware (Musca-A evaluation board) and present its applicability in an IoT water meter use case by ARM for low-power IoT devices.

The rest of the study is structured as follows. Related work is discussed in Section 1. In Section 2, we discuss in detail the technologies and background needed to understand TEE-Watchdog. Section 3 describes the threat model

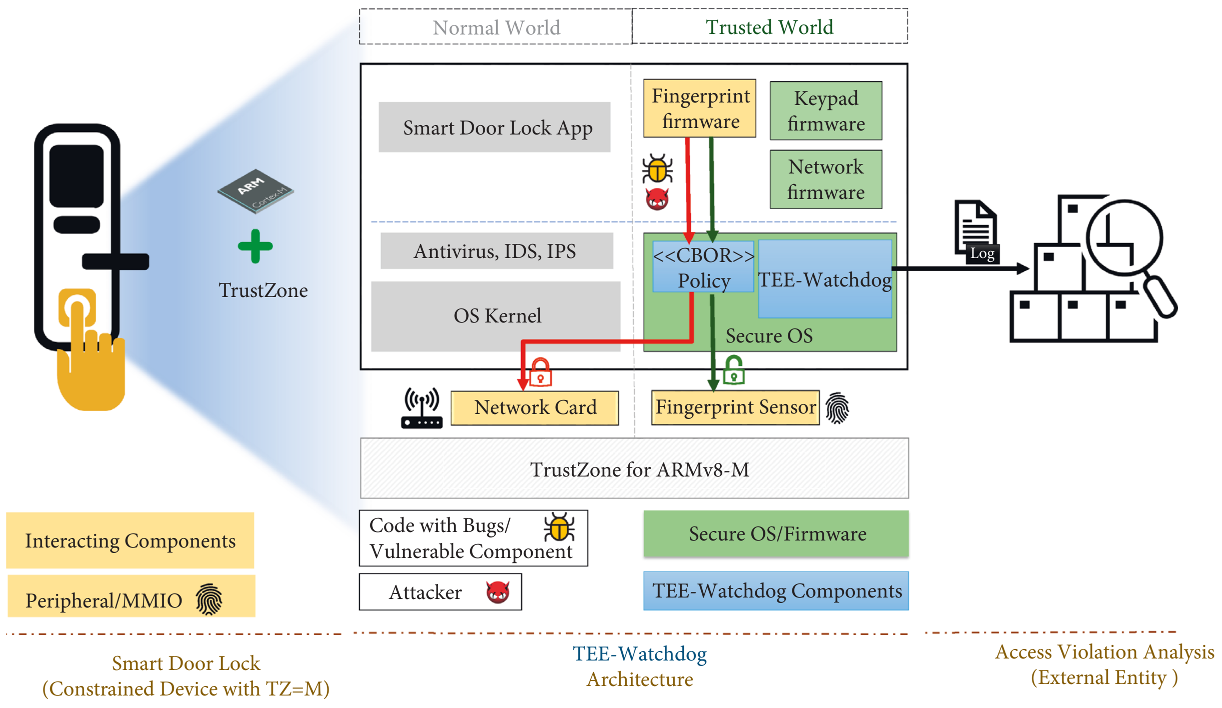


Figure 1: This figure represents the overview of the proposed mechanism and its interaction with the existing IoT system. It is proposed that peripherals (such as keypad, sensor, and fingerprint scanner) are shipped to the IoT device manufacturer together with CBOR-encoded manifest files approved and signed by a trusted authority. The IoT device manufacturer can also use self-approved signed policies. This CBOR file defines the required resources and the IoT application needs to perform its normal operation. The IoT device is configured with the policies, and TEE-Watchdog enforces the policies to protect against any malicious behaviour by the third-party software in the TEE.

followed by the TEE-Watchdog design in Section 4. The implementation details are explained in Section 5. In Section 6, we follow the implementation with the performance evaluation and results. We discuss a use case of TEE-Watchdog in Section 7. Security analysis of our proposed design is discussed in Section 8. The study concludes with Section 9.

4. Related Work

In the following section, we discuss the related literature: trusted execution environments, incidents of breaching TEEs, policy-based access of peripherals in IoT, and sandboxing IoT applications.

4.1. Trusted Execution Environments (TEEs). Trusted execution environments provide an isolated environment for software processing and execution where critical system components can be separated using virtualization or hardware-based mechanisms. Many approaches to protect code and data exist [5–7, 10–13] for embedded systems, commodity PCs, cloud environments, etc. Intel Software Guard Extensions [14] provide a platform for commodity PCs and cloud environments to set up secure isolated domains called enclaves for isolated execution of application. ARM TrustZone provides mechanisms to separate application code, system resources, and peripherals into secure and normal worlds where the critical operations execute in the secure domain and all other operations including OS operate in the normal domain. With the introduction of TrustZone-M by ARM, resource-constrained IoT devices can also create TEEs for the isolation of critical operations. Prior to the introduction of TrustZone-M, techniques such as [13, 15, 16] provided efficient ways to isolate varying levels of code in microcontrollers and IoT devices based on their criticality.

4.2. Breach of TEEs. TEEs were introduced to secure software execution and operation on critical data, and their security guarantees rely on a small TCB and strong isolation mechanisms, but recently, with the expansion of IoT domain, the number of incidents of TEE exploitation is drastically increasing. Examples of incidents of TEE breach in the past decade are Boomerang [17], Armageddon [18], and attack on Qualcomm's Secure Execution Environment [19]. Boomerang is a class of vulnerabilities that arise in TEE due to the semantic gap between the trusted and untrusted domains on a TEE-enabled device and enables a normal world attacker to take control of other resources in the normal world of higher privilege using the privileged trusted domain. In Qualcomm's Secure Execution Environment, the communication channel manager was used to exploit an integer overflow vulnerability and write to the secure memory. Armageddon demonstrates that powerful cross-core cache attacks such as prime + probe, flush + reload, and evict + reload can be used to monitor secure domain activity from the normal world. All these vulnerabilities lead to compromise of the secure domain where stealthy rootkits can be installed. Since there are no ways in the existing infrastructure to monitor the secure areas of a TEE-enabled device, it is very difficult to identify whether there is a security breach in the secure domain (as shown in [20, 21]). With the availability of TrustZone-M for resource-constrained IoT devices, these attacks become equally applicable to low-end IoT devices. As the Web of IoT expands around the user, the threat to privacy with data abuse is even greater. Hence, we propose a mechanism to monitor the access of data-generating peripherals by software in the secure world of TEE-enabled IoT.

4.3. Policy-Based Access of Peripherals in IoT. IoT devices, such as smartphones, tablets, wearables, smart-home assistants (e.g., Google Home and Amazon Echo), and wall-mounted cameras, come equipped with various sensors, notably camera and microphone. The threat is greater in a multi-vendor environment, and it becomes crucial that applications should be allowed to access only those peripherals that are a functional requirement. FlowFence [22] is a system that requires consumers of sensitive data to declare their intended data flow patterns, which it enforces while blocking all other undeclared flows. FlowFence is designed for smartphones and IoT devices with a few gigabytes of runtime memory. Similarly, Android applications have manifest files associated with their functionality enlisting the required permissions of the system sensors, peripherals, resources, etc. These manifests are utilized to specify access permissions, data flows, etc. [23]. The class of IoT devices with TrustZone-M enabled is among highly resource-constrained devices with few kilobytes of RAM. Hence, the implementation of such solutions becomes infeasible in low-end platforms. In low-end devices that have TrustZone-M support, misbehaviour by third-party applications in the secure domain is equally likely [24, 25]. Ditio [4] is a designed to enable auditing of sensor activities in modern mobile and IoT devices (in the normal world). It records sensor activity logs that can be later inspected by an auditor and checked for compliance with a given policy. It is based on a hybrid security monitor architecture that leverages both ARM's virtualization hardware and TrustZone. However, it does not provide any mechanism to block unauthorized access or violation of the compliance policies, neither does it support detection of misbehaving sensors in the secure world of TrustZone-enabled devices.

4.4. Sandboxing IoT Applications. Isolating software modules from interfering with each other, also known as sandboxing code, is a way to prevent software from affecting each other's functionality. It is utilized for preventing faults in one software affecting another, for preventing interference between applications, enabling previously approved behaviour of applications, etc. ARMor [26], ARMLock [27], and uSFI [28] are software fault isolation mechanisms applied to normal operating world, which prevent code from adversely affecting each other using hardware and software means. We propose our solution influenced by this concept of sandboxed modules in the secure world of a TrustZone-enabled device. When secure software becomes active, a subset of memory-mapped peripherals becomes visible to the software. This subset is enabled by leveraging the MPU and is in accordance with the usage policies discussed in detail in Section 4. To the best of our knowledge, our proposed solution is the first attempt that looks closely at monitoring behaviour of secure applications and automated blocking of peripheral access based on policies in manifest files. Our proposed solution can be used with any of the previously discussed code isolation mechanisms and TEEs that support TrustZone MPU-like protection and a mechanism to handle memory-managed faults.

5. Technologies Used in TEE-Watchdog

In this section, we discuss the fundamental technologies used in the design of TEE-Watchdog.

5.1. ARM TrustZone's MPU-Based Protections. Embedded systems and low-end IoT devices have strict energy budgets. These are battery-operated and usually designed to run unattended for years on the same batteries. Thus, instead of integrating full-fledged memory management units (MMUs), these systems use primitive access controllers such as memory protection units (MPUs) that provide secure and trusted execution capabilities to IoT devices [29]. MPUs are programmable blocks present in the processor that enable system developer to divide system memory (flash, RAM, ROM, MMIO) into a number of regions that can be assigned access permissions. The MPU can be configured to support 8 or 16 regions by privileged software using a series of 32 bit memory-mapped registers. The ARMv8-M architecture has 8 available MPU regions [30]. The Region Attribute and Size Register (*RASR*) is used to define the region size and memory attributes of an MPU region. The Cortex-M23 and Cortex-M33 can have up to two MPUs (one for secure world and one for normal world) if the TrustZone security extension is implemented and enabled. The secure and non-secure MPU can be configured independently with a different number of MPU regions to protect memory for the associated security domain. An MPU-based access control allows management of regions by setting (i) shareability, (ii) access permissions (read/write), and (iii) code execution permission. The MPU_RBAR register, which is a 32 bit memory-mapped register, stores the starting address of the MPU-protected region and the access permissions (Table 1 describes the register's bits). All memory accesses to that region are overseen by the MPU, including instruction fetches and data accesses from the processor, which can trigger a fault exception when an access violation is detected. As a result of these fault exceptions, the processor populates the MemManage Fault Status Register (MMFSR) and the MemManage Fault Address Register (MMFAR).

5.2. MemManage Fault Status Register. In ARMv8-M architecture with main extension, fault status registers (xFSR) are available to allow fault handlers to identify the cause of the fault exceptions. Each fault has an associated FSR. When

Table 1: MPU_RBAR bits and description of individual register bits.

Bits	Bit name	Description
[0]	XN	1 = allow code execution from this region 0 = do not allow code execution from this region
[2 : 1]	AP	Access permissions 00 = read/write by privileged code only 01 = read/write by any privilege code 10 = read-only by privileged code only 11 = read-only by any privilege code
[4 : 3]	SH	Shareability of the region 00 = non-shareable 01 = outer-shareable 10 = inner-shareable
[31 : 5]	Base	Base address of the MPU-protected region

a fault occurs, the processor pushes several CPU registers on the stack before entering the fault handler. These 32 bit registers can be inspected to further debug the cause of the fault. For MemManage faults triggered as a result of an illegal access on an MPU-protected region, the MMFSR bits can be accessed to identify the nature and cause of the fault. The MMFSR indicates the cause of the fault. By checking the values of bit 0 to 7, we can locate what type of access caused the fault. For example, if bit 0 of the MMFSR is set it indicates that an instruction fetch was attempted from a location, which did not have execution permission as per MPU configurations. Table 2 summarizes the function of various bits of the MMFSR.

5.3. MemManage Fault Address Register. The MemManage Fault Address Register (MMFAR) is also populated when a MemManage fault occurs. It contains the address of the location that generated a MemManage fault. In other words, this register is updated with the address of a location that produced a MemManage fault and can be used to retrieve the faulting address. The MMFAR address is valid only if the MMARVALID bit of MMFSR is set.

5.4. Manifest Files in IoT. Manifest files are a collection of metadata about the firmware or applications of an IoT device, including information about software location, supported devices, access policies associated with software modules regarding the accessibility of system components, and cryptographic information protecting the manifest. Manifest files are often platform-specific, and platform vendors provide application developers and vendors with a set of rules and instructions that the metadata file should comply with. Android manifests for smartphones contain access permissions about system peripherals, which the application requires access to (shown in Figure 2(a)). Android application developers have to comply with an XML-based template for the manifest file [31]. IoT firmware has associated manifest files, which are required to ensure secure over-the-air updates in zero-touch IoT devices. Ubuntu also enlists specifications [32] regarding packaging format intended to be used by third-party applications in a JSON-based manifest (Figure 2(b)). Trusted Firmware-M (TF-M), a TrustZone-M reference implementation, is under development for IoT devices [10]. Figure 2(c) shows an example of a manifest file associated with TF-M, containing details about a system partition, its priority, peripherals, etc.

5.5. Concise Binary Object Representation. Concise Binary Object Representation (CBOR) is a binary serialization data format that the Internet Engineering Task Force (IETF) proposes for IoT applications [33]. It is an ideal fit for IoT environments as it contributes to the goal of a lightweight stack for resource-constrained IoT devices. The encoders and decoders for CBOR are implemented with a small codebase. Its minimal code footprint and small message size make it suitable even for most constrained IoT devices with kilobytes of RAM. We use CBOR to implement the manifest file (discussed in Section 4.1).

6. Threat Model/Attack Model

In our *threat model,* we define and distinguish three classes of attackers. First, a trustworthy vendor could run an insecure third-party's piece of code/API in TEE unknowingly; this could be due to leaving known vulnerabilities in the code from the beginning or code becoming vulnerable over a passage of time. This allows any attacker (A1) to exploit the known vulnerabilities. Second, the vendor herself is semi-trusted for being *honest but curious* and is able to access other application data in the secure world (A2). Third, a large number of IoT devices are now being manufactured by unknown and untrusted vendors who also do not have the mechanisms to maintain a secure supply chain (https://blog.checkpoint.com/2017/03/10/preinstalled-malware-targeting-mobile-users/); such untrusted IoT vendors and/or individual component suppliers could behave maliciously to collect critical end-user data (A3).

Our *attack model* is based on a primary assumption that a service running in the secure world has autonomous access and control over the entire system resources (both normal and secure). So, a service with access to the network stack can exfiltrate data of another service over the Internet. The goal of such an attacker could be data leakage due to conflict of interest between manufacturing vendors (A2). Similarly, other attackers, such as A1 and A3, can exploit the privilege level of their code in TEE, which can access structures and functions. We also assume that the proposed structures and components (such as the access table, and log file introduced later in Section 3) can be modified and system operations (such as manifest file parsing and behaviour logging) can be interrupted by secure software to change MPU configurations and permissions of MMIO regions stored in access table. We consider that our system runs on resource-constrained IoT devices that support ARM TrustZone-M. Our security guarantees hold with the assumption that TrustZone itself is implemented correctly and no intentional flaws and bugs are introduced in it. We trust the privileged firmware and secure bootloader. We assume that stack limit registers

Table 2: Register bits of MMFSR and their function.

Bits	Bit name	Bit function
[0]	IACCVIOL	1 = the processor attempted an instruction fetch from a location that does not permit execution
[1]	DACCVIOL	1 = the processor attempted a load or store at a location that does not permit the operation
[3]	MUNSTKERR	1 = unstack for an exception return has caused one or more access violations
[4]	MSTKERR	1 = stacking for an exception entry has caused one or more access violations
[7]	MMARVALID	1 = MMAR holds a valid fault address

are used appropriately to prevent stack manipulation and the nonsecure interrupt service routines (ISRs) cannot interrupt secure ISRs. Moreover, correct usage of stack limit registers ensures that secure ISRs are handled by privileged software. Moreover, we assume there are no validation bugs in the secure software that can lead to privilege escalation to the privileged firmware level.

Based on the above threat model and assumption, we design TEE-Watchdog with the following security goals, which will be analysed in detail in security analysis section (Section 6): (i) secure applications cannot modify TEE-Watchdog components and structures, i.e., the access table, log file, and manifest file (*G1*); (ii) normal world applications and their trusted code cannot interrupt critical system processes that make TEE-Watchdog functional (*G2*); and (iii) malicious applications are prevented from depleting TEE-Watchdog resources such as the log file (*G3*). Table 3 summarizes these identified goals.

7. Tee-Watchdog Design and Architecture

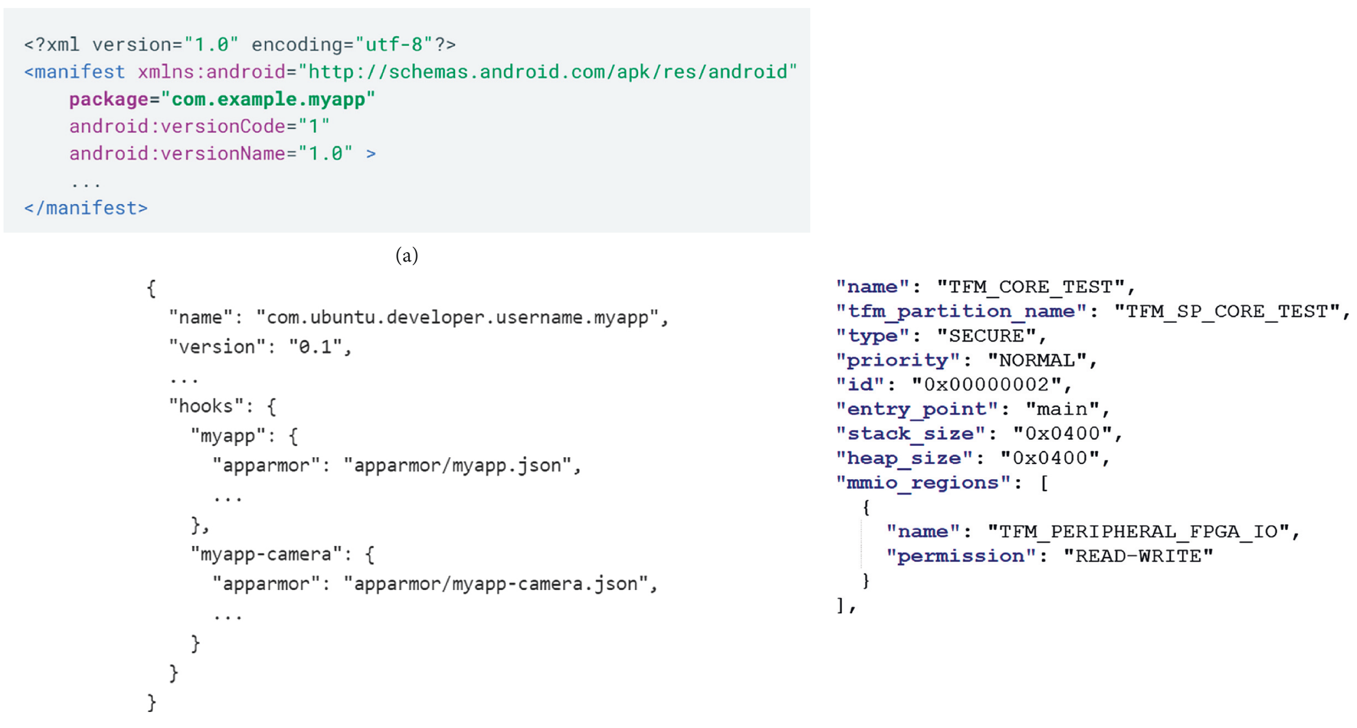
We propose TEE-Watchdog, a sandboxing and behaviour logging mechanism for secure software in TrustZone-enabled platforms. The high-level architecture of TEE-Watchdog is illustrated in Figure 3 and depicts 3 entities: an *IoT device*, an *IoT device vendor,* and an *external audit service*. We introduce the *Security Manager* as a part of the secure kernel, which is privileged secure software. It is the supervising component that handles all sub-modules. The application/service vendors provide software's *manifest file* enlisting functional details about the software, its sub-modules, and required access to system peripherals. The software can be any user-level application or firmware of a peripheral. This *manifest file* is converted by the policy converter to a memory-mapped *access table* at system boot that contains the access permissions associated with each software. The permitted peripherals are recorded in the *access table* where each application is listed along with its set of permitted peripherals. All system peripherals besides those enlisted are not permitted by default.

In IoT devices with TrustZone-M enabled, the system designer can partition the system memory into secure world and normal world. This partitioning is enabled by configuring memory maps using an Implementation Defined Attribution Unit (IDAU) and a Security Attribution Unit (SAU) at boot time. An MPU is utilized to implement a runtime access control on system memory and peripherals. MPU enforces access control on all memory accesses, including regular memory and device's memory-mapped I/O (MMIO). We leverage MPU to limit access of secure software on secure resources. The *sandboxing module* enforces MPU-based protection on secure resources and peripherals leveraging the secure *MPU*. The *audit module* intercepts MemManage faults triggered on illegal accesses by software on a secure resource to record the app and the violation that occurred, in the *log file*. We also propose a template for an application manifest file based on CBOR file format. The proposed framework requires every IoT application to come equipped with a manifest file in order to benefit from TEE-Watchdog protections. In the following sections, we discuss the system structures of the TEE-Watchdog framework and the main modules of our *Security Manager*.

7.1. System Structures

7.1. System Structures. In this section, we discuss the specifications, structures, and placement of TEE-Watchdog components, namely the *manifest file*, the *access table,* and the *log file*.

7.1.1. Application Manifest File. As IoT devices are diverse in nature with regard to the computational capabilities (i.e., network capacity, processing power, energy consumption of the IoT device, and memory capacity), the manifest file should be concise enough to be used in most constrained environments. We design the manifest file associated with secure software to be lightweight and interoperable since the proposed framework is targeting resource-constrained devices. We propose these specifications based on CBOR. Although it is less human-readable, since IoT devices are usually deployed and expected to operate with minimal human interaction, this does not constitute a problem. The availability of CBOR in off-the-shelf IoT devices and its efficiency to encode and run on low-powered devices are a strong rationale for this selection [34]. We identify some attributes that a manifest file should have to enable TEE-Watchdog to monitor secure software and log malicious behaviour: (a) a unique identifier (UniqueID) that is used to globally identify the application across platforms and vendors, (b) a list of *peripherals* that it requires for normal functionality, and (c) the respective *access permission* for each peripheral. Optional attributes including system memory requirements and security keys can also be mentioned but are not required. Application identifiers that identify service layer objects and logical entities are used across devices, platforms, and vendors to monitor the behaviour of applications. Such identifiers usually follow standard formats such as IEEE 64-bit Extended Unique Identifier (EUI-64), Uniform Resource Identifier (URI), or Uniform Resource Locator (URL) [35]. The unique identifier



```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp"
    android:versionCode="1"
    android:versionName="1.0" >
    ...
</manifest>
```

(a)

```
{
  "name": "com.ubuntu.developer.username.myapp",
  "version": "0.1",
  ...
  "hooks": {
    "myapp": {
      "apparmor": "apparmor/myapp.json",
      ...
    },
    "myapp-camera": {
      "apparmor": "apparmor/myapp-camera.json",
      ...
    }
  }
}
```

(b)

```
"name": "TFM_CORE_TEST",
"tfm_partition_name": "TFM_SP_CORE_TEST",
"type": "SECURE",
"priority": "NORMAL",
"id": "0x00000002",
"entry_point": "main",
"stack_size": "0x0400",
"heap_size": "0x0400",
"mmio_regions": [
  {
    "name": "TFM_PERIPHERAL_FPGA_IO",
    "permission": "READ-WRITE"
  }
],
```

(c)

Figure 2: Examples from various manifest files in operating systems, IoT, firmware, etc.(a) Manifest file associated with TF-M secure services indicating identifiers, priorities, access to MMIO, etc. (b) JSON-based manifest file in Ubuntu containing information for third-party applications. (c) Manifest file associated with android applications.

in the proposed *manifest file* serves the purpose of global identification of secure software. The following listing is an example of a manifest file as per the proposed guidelines.

```
{  "UniqueID"   : "AD-4E-22-C5-61-FF-AF", // Mandatory
   "Policies"   : {                      // Mandatory
                  "Temp-Sensor" : "RO",
                  "FP-Reader"   : "RW"
                  },
   "Stack-Size": "0x0400"                // Optional }
```

The CBOR encoding of the above JSON file is shown as follows.

```
A3                                        # map(3)
   68                                     # text(8)
      556E697175654944                    # "UniqueID"
   74                                     # text(20)
      41442D34452D32322D43352D36312D46462D4146 # "AD-4E-22-C5-61-FF-AF"
   68                                     # text(8)
      506F6C6963696573                    # "Policies"
   A2                                     # map(2)
      6B                                  # text(11)
         54656D702D53656E736F72           # "Temp-Sensor"
      62                                  # text(2)
         524F                             # "RO"
      69                                  # text(9)
         46502D526561646572               # "FP-Reader"
      62                                  # text(2)
         5257                             # "RW"
   6A                                     # text(10)
      537461636B2D53697A65                # "Stack-Size"
   66                                     # text(6)
      307830343030                        # "0x0400"
```

Table 3: A summary of TEE-Watchdog's security goals.

Security goal	Description
G1	Secure world applications cannot modify TEE-Watchdog components and structures.
G2	Normal world applications and their trusted code cannot interrupt TEE-Watchdog operations or processes that make TEE-Watchdog functional.
G3	Malicious applications are prevented from depleting TEE-Watchdog resources.

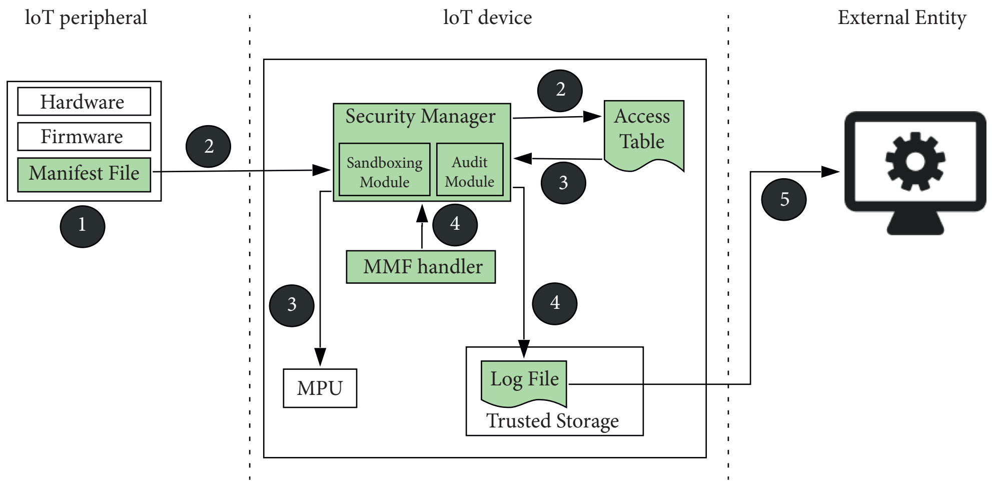


Figure 3: This figure shows TEE-Watchdog's high-level architecture and processes in the secure world of a TrustZone-enabled IoT device. (1) An IoT vendor supplies a signed manifest file along with a peripheral/sensor containing access requirements for each secure peripheral in the IoT device. (2) TEE-Watchdog's Security Manager parses the manifest file and generates an access table based on the manifest file at system boot. (3) When secure software becomes active, the Security Manager configures secure peripherals according to the permissions specified in the access (Table 4). If and when secure software tries to access peripheral beyond its access permissions, TEE-Watchdog's Security Manager fetches all information regarding the access violation and logs the event in the log file in trusted storage. (5) The log file can then be sent to an external auditor or used in training of intrusion detection systems or simply report the misbehaving software.

TEE-Watchdog's manifest file should comply with the guidelines below:

(i) The manifest file should contain a globally unique application identifier, *UniqueID*.

(ii) The *UniqueID* should be constructed by concatenating the organizational identification of the vendor and local identification assigned to the application by vendor (privately known to the vendor).

(iii) Standards such as ISO/IEC 6523-1 [36] and IEEE-administered organizational identifiers, organizationally unique identifier (OUI) and company ID (CID) [37], are used to assign unique identifiers to organizations and companies. The OUI can be extended to add a local identifier assigned to identify applications. We propose using EUI-64 [37] for globally identifying the application software as they are intended to be used by applications that require globally unique identifiers for interfaces or instances.

(iv) Following is a hexadecimal representation of a *UniqueID* complying with EUI-64 comprising of octets separated by hyphens.

(v) "UniqueID": "AD-4E-22-C5-61-FF-AF."

This *UniqueID* is constructed by adding additional bits to an IEEE-administered OUI or CID (e.g., OUI-36 and CID-24). In the above example, the first (most significant) four and half octets are the OUI-36 and the remaining hexadecimal values are the unique application identification assigned by the vendor to construct an EUI-64.

(i) The manifest file should contain a list of *policies* specifying the type of access required by the software for every secure system peripheral. The platform peripherals not mentioned in the manifest file are by default not accessible to the software.

(ii) The content of the manifest file should be an array/map containing name-value pairs complying with the CBOR encoding format, for example.

(iii) "Temp-Sensor": "RO."

(iv) The names of peripherals specified in the manifest file should be the same as enlisted by the IoT device vendor (usually in the datasheet or specifications) to maintain uniformity across devices.

(v) Access permissions for the system peripherals should be either "RO" for read-only access or "RW" for read-write access.

The attributes of the manifest file are proposed in a minimal size considering the diversity in memory availability of IoT devices of different classes.

7.1.2. Access Table. An access table is part of the proposed framework and is generated during system boot time. It enlists the secure software installed on device and stores their access permissions regarding the system peripherals (shown in Table 4). Section 7.2.2 describes the process of converting manifest file to platform-specific access table.

7.1.3. Log File. The log file is another major component in the proposed mechanism. It is populated only by the Security Manager and stored encrypted as a secure system resource. A log file containing information about behaviour violations is shown in Table 5. Every violation is added with a (i) *Violation Code* depicting the kind of illegal access performed by the application, (ii) *UniqueID* of the application that caused the violation, (iii) *Violated_Peripheral,* and (iv) *Address* that is being accessed during the violation.

7.2. Sandboxing Module. The sandboxing module of our Security Manager deals with the verification of manifest file, its translation to the access table, and enforcement of the access policies from the manifest file. It consists of submodules: verifier, policy converter, and policy enforcement module.

7.2.1. Manifest File Verification. The lifecycle of a manifest file associated with the software, containing metadata about the hardware, software, or firmware, begins with the vendor specifying critical attributes in the file. In our proposed system design, we assume the IoT component vendor would provide the manifest file along with the hardware to the device manufacturer. It is possible for the IoT device manufacturer to manufacture its own parts, in which case it also becomes responsible for specifying the policies. The policies of a sensor/peripheral stated in the manifest file would be publicly available as a resource. The device manufacturer is responsible for the integration of the individual components and assembling the IoT device. It also loads the software and manifest file on the device. The device is then shipped to the user as a solution ready for deployment. The IoT device is a package of (i) the hardware, (ii) the firmware and its certificate, (iii) the integrated peripherals along with their software drivers, and (iv) the manifest file with an encrypted unique identifier and hash/certificate of the manifest file. The *device config file* is a trusted component present in current state-of-the-art IoT devices. It contains device configuration details and can include the paths to the secure firmware and the manifest file. During device deployment, the *device config file* is configured and the manifest file of each software is loaded. After secure boot has transferred the control to the privileged secure kernel, the *verifier* starts loading each manifest file for access table generation. Each manifest file is hashed and verified against the preinstalled list of legitimate manifest file hashes. Once it is established that a manifest file was not modified or replaced, the policy converter module is invoked.

7.2.2. Policy Converter. Our system design introduces a *policy converter* as a part of the system boot process. During device bootup and on system update, the *Security Manager* checks whether there is a manifest file associated with a secure application. A memory-mapped *access table* is created during this step, which contains a list of applications installed and their access permissions for all secure system peripherals. Given a list of system peripherals and a list of access policies extracted from the manifest file, the policy converter generates an access table (represented in Table 4) mapped to the MMIO. Algorithm 1 is a representation of the interpretation steps that the policy converter performs. For each application, the policy converter identifies the peripherals from the provided list of peripherals and enters their addresses to the access table. These peripherals are marked in the access table as permitted, and all other peripherals are nonaccessible for the specific software.

7.3. Policy Enforcement. As previously discussed, Cortex-M23 and Cortex-M33 have up to two MPUs. We leverage the secure MPU to protect system resources in the secure world including code, data, MMIO regions, and other system structures. When a secure service is invoked by a normal world application, it becomes the currently active application/service in the secure world. Based on this active service in the secure side, the policy enforcement module configures the entire secure memory space as per the *access table* (Algorithm 13, 2). If a resource in the access table has permissions listed against a specific service, the *policy enforcement module* protects the resource using an MPU and assigns the given permissions to the specific memory region. As a result, the secure service can only access system resources and peripherals as per the access table. Figure 4 depicts the process of policy enforcement in a step-by-step way. The integrity of the secure MPU and the access table is critical for the proposed system architecture. As they are system resources located in the secure memory, their integrity is guaranteed using MPU protections. The access permissions for each of these structures are configured to be editable only by privileged secure software, which is our Security Manager.

7.4. Audit Module. We propose an audit module as a part of our Security Manager, which is responsible for overseeing the behaviour of secure software regarding peripherals and resources being accessed.

7.4.1. Behaviour Logging of Applications. The audit module is activated only when a service/application behaves outside its specified permissions and attempts to access memory or peripherals beyond a permitted list. Prior to this, the secure memory is divided into MPU-protected regions by the sandboxing module with permissions configured according to the access table. During such a violation, the service making an illegal access to a secure peripheral would trigger a

Table 4: Platform-specific access table generated from manifest file at system boot.

UniqueID (application identifier)	Peripheral (peripheral address)	AP (access permission)
9A-49-32-8A-32-BF-44	0 × 40000000	ReadWrite
	0 × 50000000	ReadOnly
	0 × 60000000	ReadOnly
AD-4E-22-C5-61-FF-AF	0 × 40000000	ReadOnly
	0 × 50000000	ReadOnly
	0 × 60000000	ReadOnly
DA-4E-22-C1-67-1F-DF	0 × 40000000	ReadOnlyPriv
	0 × 50000000	ReadOnlyPriv
	0 × 60000000	ReadOnlyPriv

Table 5: Log file, populated with records of violation details.

Violation	Details		
Code	*UniqueID*	*Violated_Peripheral*	*Address*
RW	AD-4E-22-C5-61-. . .	Fingerprint_Scanner	0 × 50000000
XN	CD-4E-82-35-61-. . .	Gyro sensor	0 × 70000000
. . .	. . .	. . .	0 × 7. . .
. . .	. . .	. . .	0 × 7. . .

MemManage fault. As soon as the processor triggers the fault, it populates the MMFAR with the memory address of the resource being accessed and sets the MMARVALID bit in the MMFSR indicating that MMFAR holds a valid fault address and sets the bit (0 to 7) in MMFSR corresponding to the type of access that generated the fault. We modify the MemManage fault handler to introduce the audit module. The audit module checks the LogFileExists flag before continuing to investigate the fault. If the MMARVALID bit is set, the audit module reads the address from MMFAR (as shown in Figure 5) and enters it into the log file (Table 5) along with the details of the violation and the application's UniqueID. The log file suffers the risk of being overpopulated since the device memory is constrained. At any point, the log file can only maintain a limited number of violation records. To ensure that all records of violation are stored, the audit module checks the number of existing log entries before making a new entry. If it is equal to maxEntries, the existing entries of log file are moved for further processing to an external audit service and new entries can be stored on device. The Security Manager maintains a list of secure services and sets or clears the isActive flag based on the service that was invoked from the normal world or by another secure service. This ensures the link between the active applications that caused the violation and the log file entry. After the attempt is successfully logged, the control flow returns to the MemManage fault handler and the fault is handled as per system settings. Algorithm 3 depicts the procedure, which handles log file entries.

7.5. Usability of Log File. The log file is stored in MPU-protected regions in flash or trusted storage provided by the hardware to ensure integrity. The MPU region where the file is stored is configured to allow only our secure privileged Security Manager to access or modify the contents of the log file. The log file is structured to contain sufficient information about the access violations, which can be evaluated in multiple ways. It contains the UniqueID of the violating entity in the current system, which is globally unique as described in Section 4.1. The log file is intended to be shareable to external entities for further processing, and the UniqueID can sufficiently identify the misbehaving application and the vendor. There are multiple ways the log file can be processed to determine the course of action for policy-violating software. The misbehaviour of service/applications is diversion from the initially agreed policies. One suitable option is disabling the service on device until a software update fixes the problem. It is likely that such an access pattern could arise from a wrongly configured policy, but it is also equally possible that this behaviour is intended by vendors for industrial espionage. If the violations show a pattern across devices, these increasing numbers of violations from an application on multiple IoT devices can be made public and reported back to the vendor, and this forces the vendors to patch the problem. Besides this, it also allows IoT devices without TEE-Watchdog support to identify such applications and vendors before making installations on their devices.

A more sophisticated use of such a log file would be as an input to an intrusion detection system as part of a threat intelligence platform. The Cyber Threat Intelligence Technical Committee (CTI TC) of the Organization for the Advancement of Structured Information Standards (OASIS) proposes standards that facilitate the exchange of threat information. Structured Threat Information eXpression (STIX) is one such standard that allows automatic information exchange between multiple tools [38]. STIX is based on JSON, and its objects represent *indicators*, *malware*, and relations between the objects. The information from the log file such as *UniqueID* and *Violation Code* could be converted to STIX-understandable objects representing an *indicator* or a potential *malware*.

8. Implementation

TEE-Watchdog'*s* design was based on ARMv8-M [10], which is a 32-bit ARM architecture for Cortex-M processors.

8.1. Runtime Environment. The implementation of TEE-Watchdog prototype builds on Trusted Firmware-M (TF-M) in the secure side with CMSIS RTOS2 (https://www.trustedfirmware.org/) as normal world OS. Approximately

```
Input: Enco de d_Manif est_File, Peripherals     ▷ CBOR-encoded Manifest File and list of system peripherals
Output: Access Table
procedure Policy_Converter Enco de d_Manif est_File, Peripherals
Policy_File = Decode Enco de d_Manif est_File
3 Initialize Access Table                        ▷ Initialize the Access Table data structure
i ⟵ 0
token = Tokenize Policy_File                     ▷ Begin converting text into tokens and get first token
while token do
  if token = = "UniqueID" then
    token = Tokenize Policy_File                 ▷ Get next token
    Access Table [i].app_ID = token
  else if token = = "Policies" then
    while 1 do
      token = Tokenize Policy_File               ▷ Get next token
      ti←0
      while ti is less than sizeof (Peripherals) do
        if token = = Peripherals[ti].name then
          Access Table [i].peripheral = Peripheral[ti].addr
          token = Tokenize Policy_File           ▷ Get next token
          if token = = "RW" then
            Access Table [i].perm = = "RW"
            i ++
            break
          else if token = = "RO" then
            Access Table [i].perm = = "RO"
            i ++
            break
          else
            i ++
            break                                ▷ Invalid permission
          end
        else
          ti ++
        end
      end
      if token has terminator then
        break
      end
    end
  else
    do nothing
  end
end
end procedure
```

Algorithm 1: Translating the application's manifest file to memory-mapped access table

460 lines of code were written to include TEE-Watchdog components in the firmware. TF-M provides a reference implementation of secure world software for ARMv8-M [10]. It creates the foundations of TEE by providing a set of secure runtime services such as secure storage, cryptography, and attestation. Additionally, secure boot in TF-M ensures the integrity of runtime software and supports firmware upgrade.

8.2. TEE-Watchdog Components. We discuss the implementation details of TEE-Watchdog modules in the runtime environment discussed above.

8.2.1. Manifest File Translation. The policy converter module converts the policies associated with the application about the usage of MMIO into a memory-mapped access table. The manifest file is encoded in CBOR when received accompanying the application. We implement the policy converter using the QCBOR decoder (https://github.com/laurencelundblade/QCBOR) for RFC 7049. The decoded manifest file is then parsed to extract the attributes, which are used to populate the access table. The parsing algorithm is represented in Algorithm 1. We implemented access table as a system structure stored in the secure world. It consists of a list of applications that need to access data from secure peripherals and the access

```
Input: Access Table, ActiveApp
Output:
success_code
Procedure Policy_EnforcementAccess Table, ActiveApp
success_code ⟵ −1
index ⟵ 0
while index ≠ Size_of(Access Table) do
    If Access Table[index] = = ActiveApp
        region_a dd r ⟵ Access Table
        [index].
        region_add r
        AP ⟵ Access Table [index]. AP
        Set_Protection
        (region_add r, AP)
                                                        ▹ Call Procedure
        index ++
        success_code ⟵ 0
                                                        ▹ The procedure is successful
    end
end
Return success_code;
End Procedure
```

Algorithm 2: Enforcing the policies specified in the access table

```
Input: region_addr, AP                          ▹ Address of the region and permissions
Output:
success_code
Procedure Set_Protection
region_addr, AP
success_code ← − 1 Set bit 0 of MPU_CTRL to 0                    ▹ Disables MPU
Set bits [31:5] of MPU_RBAR to region_addr          ▹ Sets region address to be protected
if AP == ReadWritePriv then
    Set bits [2:1] of MPU_RBAR to 00    ▹ Sets Access Permission of region_addr to be Read/Write by privileged code only
else if AP = = ReadWrite then
    Set bits [2:1] of MPU_RBAR to 01 ▹ Sets Access Permission of region_addr to be Read/Write by any code
else if AP = = ReadOnlyPriv then
    Set bits [2:1] of MPU_RBAR to 10 ▹ Sets Access Permission of region_addr to be Read Only by privileged code only
else if AP = = ReadOnly then
    Set bits [2:1] of MPU_RBAR to 11      ▹ Sets Access Permission of region_addr to be Read Only by any code
else
    Set bit 0 of MPU_CTRL to 1              ▹ Enables MPU protection for the region
    success_code ← − 1
    return success_code
end
Set bit 0 of MPU_CTRL to 1                                   ▹ Enables MPU protection for the region
success_code ←0                                                    ▹ The procedure is successful
return success_code;
End Procedure
```

Algorithm 3: Enabling MPU protection using the Procedure Set_Protection

permission against each system peripheral as specified by the manifest file. Before the translation of manifest file begins, the policy converter module confirms the authenticity of the manifest file. We used SHA-512 provided by TF-M crypto-services to calculate the list of manifest file hashes.

8.2.2. Access Table Enforcement. The policy enforcement module in our proposed scheme uses the access table generated during boot time by the policy converter. We implemented the policy enforcement module to utilize the secure MPU using the *mpu_armv8m_drv API* provided as a part of Trusted Firmware-M and configure all MMIO in

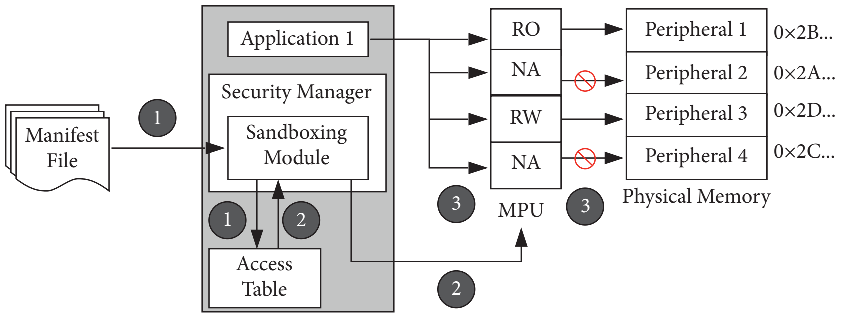


Figure 4: (1) The sandboxing module of the Security Manager translates the application's manifest file into system-specific access table, (2) when an application becomes active, the sandboxing module configures the secure peripherals according to the access table and enforces MPU protections, and (3) the application can then only access peripherals according to the permissions.

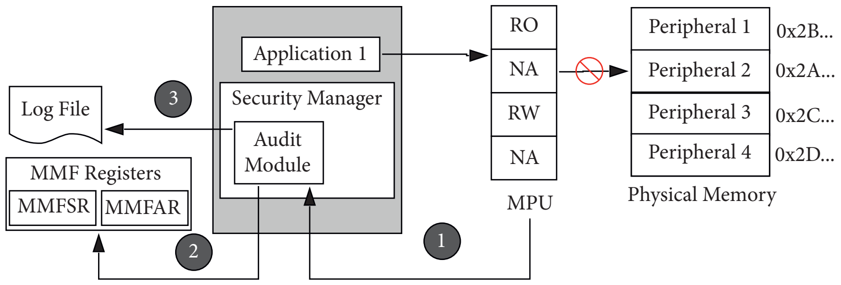


Figure 5: The audit module of the Security Manager performs behaviour logging of application deviating from their intended resource access: (1) when a software attempts an illegal peripheral access, a MemManage fault is triggered, which invokes TEE-Watchdog's audit module, and (2) the audit module reads from the MemManage fault register to locate the violation and (3) populates the log file based on the active application and the type of access violation.

```
Input: ActiveApp                               ▷ Application ID causing the access violation
Output: success_code
Procedure Log_Violation(ActiveApp)
ActiveAppsuccess_code ← − 1
Initialize Log_Entry          ▷ A datastructure representing Log File entry is initialized with null values
if bit 7 of MMFSR == 1 then                           ▷ The MMFAR holds a valid fault address
    Log_Entry.App_I D←ActiveApp                              ▷ Violating application
    Log_Entry.Violate d_Peripheral← MMFAR                   ▷ Violated peripheral address
    if bit 0 of MMFSR == 1 then
        Log_Entry.Violation_Code ← XN                        ▷ Code execution attempt
    else if bit 1 of MMFSR == 1 then
        Log_Entry.Violation_Code ← RW                         ▷ Read or write attempt
    else if bit 3 of MMFSR == 1 then
        Log_Entry.Violation_Code ← ER              ▷ Access violation due to exception return
    else if bit 4 of MMFSR == 1 then
        Log_Entry.Violation_Code ← EE               ▷ Access violation due to exception entry
    else
        Log_Entry.Violation_Code ← UE                                        ▷ Unknown fault
    end
    Write Log_Entry to Secure Storage
    success_co de←0                                          ▷ The procedure is successful
else
    success_code ← − 1;
                          ▷ MMFAR does not contain a valid address and Log_Entry is not added to Log File
end
return success_code
End Procedure
```

Algorithm 4: Creating a log entry of access violation into log file

accordance with the access table. The *mpu_armv8m_drv API* uses the *MPU_BASE* address to access the MPU for configurations.

8.2.3. Policy Violation Logging. We implemented the *MemManage fault handler* to retrieve information from the *MMFAR* and *MMFSR* registers about the memory access violation. We use the *tfm_sst_api* to write the log file entries to the log file. The log file is stored in the TFM-SST (TF-M service for storing sensitive data). TF-M provides 20 kb of secure storage where data are stored encrypted using AEAD encryption (with a fresh nonce for each encryption) with a hardware unique key (HUK).

9. Evaluation

In this section, we critically evaluate the performance of TEE-Watchdog prototype. We perform a set of microbenchmarks to an application accessing the temperature sensor placed in the secure side as a secure peripheral. We select CPU time, which is directly proportional to energy consumption as our performance metrics. Memory and energy consumption are the most constrained resources in low-power IoT devices. The quantified CPU time is directly proportional to the power consumed by the TEE-Watchdog operations. We also calculate the end-to-end-latency of a simple temperature monitoring IoT application and measure the delay in latency due to TEE-Watchdog protection mechanisms. We compare our CBOR-encoded policy to JSON file format and highlight the reduction in file size. We also discuss the minimal impact on RAM availability due to TEE-Watchdog services establishing the lightweight nature of the proposed scheme.

9.1. Experimental Setup. We evaluated the performance of TEE-Watchdog on the Musca-A2 test chipboard by ARM shown in Figure 6. The Musca-A2 board implements the ARM CoreLink SSE-200 subsystem featuring dual-core Cortex-M33 with CPU0 enabled at 50 MHz. We use TF-M and CMSIS RTOS v2 enabled with TEE-Watchdog components for these experiments. The execution time is measured using the CoreSight debug port found on Musca-A2 test chipboard. The CoreSight debug port contains a 32-bit free-running counter that counts CPU clock cycles. The counter is part of the debug, watch, and trace (DWT) module, which we use to measure the execution time of our code.

9.2. Comparison of CBOR Vs. Other File Encoding Formats. We implement manifest file in both CBOR and JSON. Our comparison in Table 6 clearly shows that representing the same number of policies in CBOR shows an average of 40.81% reduction in file size as compared to JSON.

9.3. Memory Overhead. The Musca-A2 IoT evaluation chip has 256 kb of RAM available. The runtime impact of TEE-Watchdog on system RAM was 1.79 kb, which reduced the RAM availability merely by 0.7%

9.4. Performance Evaluation of TEE-Watchdog Components. We evaluate the system from two dimensions. In the first part of our evaluation, we analyse the individual modules and mechanisms of our proposed solution. We measure the execution time in terms of CPU time of TEE-Watchdog mechanisms that enable us to provide secure peripheral protection and behavioral logging.

9.4.1. Populating the Access Table Based on Manifest File. During system bootup, the CBOR-encoded manifest file associated with an application is decoded as explained in Section 5 and the policies are then parsed to an access table. Before the manifest file can be parsed, our Security Manager verifies the authenticity of the manifest file against a previously computed set of hashes. This procedure is described in detail in 4.2.1. We perform the evaluation on manifest file containing up to 8 policies to estimate the maximum delay incurred on system bootup time due to this step. Authenticating a single manifest file depending on the number of policies is shown in Table 7. Translating a decoded manifest file adds a delay of approximately 298.34 microseconds in CPU time for a manifest file with 2 policies (as shown in Table 7). We find the total overhead on the system bootup due to (i) manifest file verification, (ii) decoding from CBOR, and (iii) translation into platform-dependent access table to be 1312.96 microseconds for a manifest file with 2 policies. We find this overhead to be justified based on the fact that system restart in embedded IoT devices is a rare event and does not impact the real-time functionality of the device.

9.4.2. CPU Time to Setup TEE-Watchdog Protections Based on Access Table. When the application requests a secure service to invoke a secure peripheral, the control is switched to the secure side and TEE-Watchdog configures all the secure peripherals according to the access table. At this moment, only the peripherals that are allowed to be used by the application are accessible to be read. When the process is complete, the control flow is redirected to the normal world, but before doing so, all the memory configurations are reset to the default configurations. In our evaluation, we find the time to enable and disable the memory-mapped protections to be 47.6 microseconds and 13.7 microseconds, respectively, with one secure peripheral. Figure 7(a) shows the CPU time for execution of enabling and disabling TEE-Watchdog protections based on the number of secure peripherals in the system. Considering that resource-constrained IoT devices usually have fewer peripherals as these devices are built to be function-specific, a delay of 156 microseconds to enable protections for 4 secure peripherals is minimal overhead. The impact of this overhead on IoT applications is discussed in later sections.

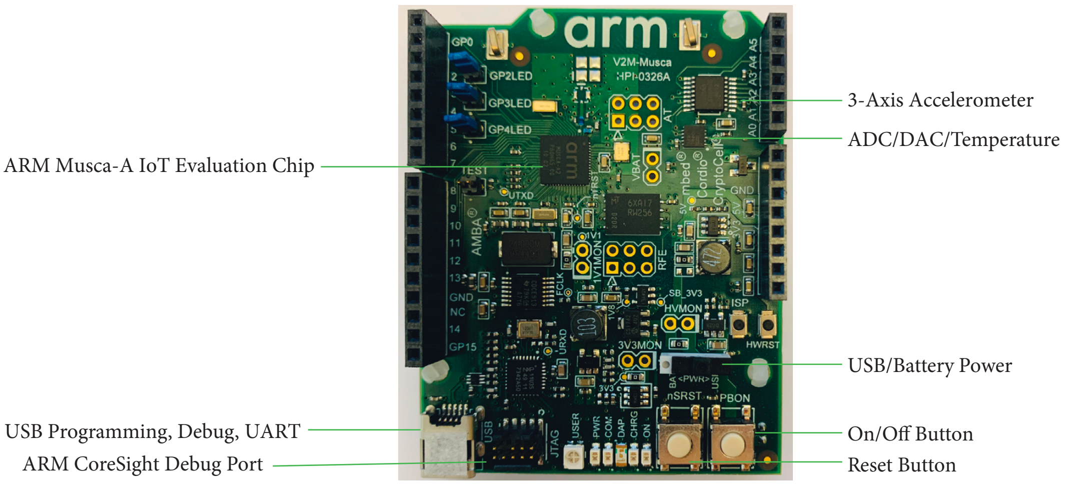


Figure 6: ARM MUSCA-A test chipboard based on ARM Cortex-M33 processor.

Table 6: A comparison of size of manifest file defined in CBOR vs. JSON. The table shows the reduction in file size by an average of 40% if CBOR encoding is used to encode the manifest file.

No. of policies	Size of manifest file in JSON (bytes)	Size of manifest file in CBOR (bytes)	Reduction in size using CBOR (%)
1	119	74	37.8
2	144	87	39.5
3	172	103	40.1
4	199	118	40.7
5	224	131	41.5
6	246	141	42.6
7	278	159	42.8
8	306	177	42.15

9.4.3. CPU Time to Log an Access Violation of Policies by an Application. When the memory-mapped protections of secure peripherals are enabled, any illegal access will generate a MemManage fault. Our modified fault handlers then collect the information about the memory access violation and create a log entry with the following details: (i) *Violation Code* indicating memory-read or memory-write violation, (ii) *UniqueID* belonging to the secure application that tried to access memory against its permissions, and (iii) *Violated_Peripheral* and *Address,* which is the violated peripheral address (as discussed in Section 7.1.3). This log entry is then written to the secure storage against the name of the application that is violating the access policies. The CPU time to make one entry to the log file is 50.16 milliseconds. This is the most time- and energy-consuming task of the TEE-Watchdog protection mechanism and is dependent on the size of the log file entry that needs to be written to secure storage.

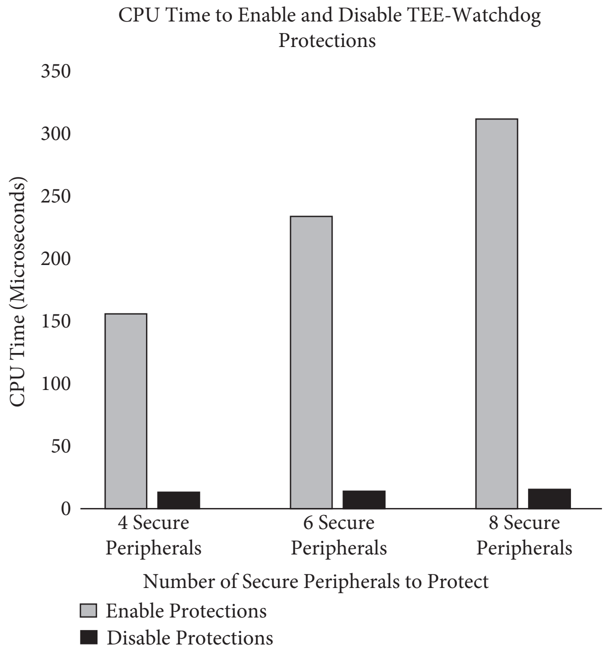
9.5. TEE-Watchdog's Overhead on End-to-End Latency of Applications. The second part of our evaluation is targeted at deducing the impact of TEE-Watchdog protection mechanism on the existing system applications. We perform this set of experiments to comprehend the delay in execution or overhead caused by TEE-Watchdog on peripheral accesses. We measure the average CPU time of execution of a peripheral access by an application without the presence of TEE-Watchdog security mechanism, which is our baseline case. After enabling TEE-Watchdog protections, we find that there is 1.41% overhead on each peripheral access due to policy enforcement module configuration of secure peripherals every time the control is transferred to the secure world. The average delay in execution/latency is a negligible 61 microseconds if one secure peripheral exists in the system. Figure 7(b) shows that there is an increase in the latency with the increase in the number of securely protected peripherals. The latency increases from 1.4% to 8.8% if the number of protected peripherals increases from 1 to 8. Resource-constrained IoT nodes are usually function-specific, designed with few peripherals and applications. In application scenarios where the application takes a reading after timed intervals rather than in a real-time mode, the delay in execution does not impact any critical system operation.

10. Water Meter—An ARM PSA-Compliant Use case

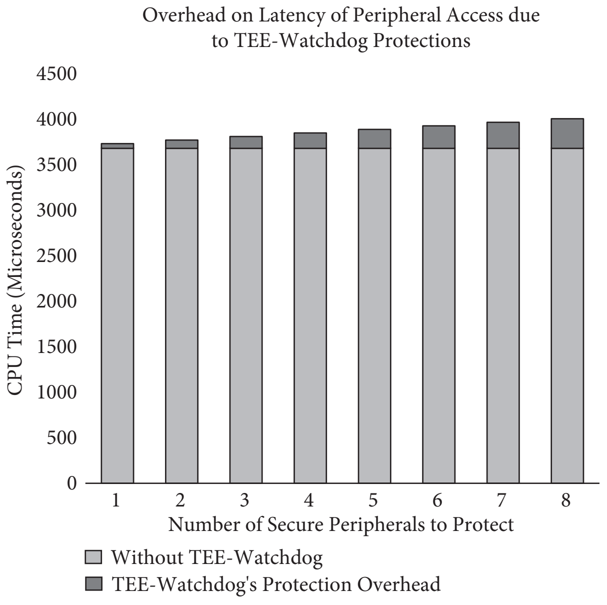
Water meters are among the constrained class of IoT devices deployed on massive scale and operating on batteries for years to reduce the cost of maintenance. Such water meters are owned by water distribution companies targeting homes, offices, industries, or farmlands. Industrial or farmland meters differ from standard home/office-based meters in the

Table 7: CPU time in microseconds for manifest file verification and manifest file translation into platform-dependent access table

TEE-Watchdog procedures during boot time	Manifest file details		
	49 bytes *No policies*	*87 bytes* *2 policies*	*118 bytes* *4 policies*
Manifest file verification	667.52 μ S	668.28 μ S	1035.88 μ S
Manifest file translation	208.04 μ S	298.34 μ S	415.32 μ S



(a)



(b)

Figure 7: This figure shows two trends. (a) The trend on the left is based on the increasing number of secure peripherals present in the system that require configuration and show the CPU time to enable and disable TEE-Watchdog. (b) The second trend shows the overhead of TEE-Watchdog protection mechanisms on latency (measured as delay in CPU time in microseconds) of a peripheral access. The latency increases from 1.4% to 8.8% based on the increasing number of total secure peripherals that are protected in the system.

number of functionalities. In this section, we discuss the deployment and lifecycle of a *water meter* as a use case of TEE-Watchdog. *Water meters* are part of a huge network of data-collecting infrastructure. Breach of such devices results in incorrect water supply measurement, overcharging the billing amount, and blockage of water supply in worst-case scenario. We identify operational and security requirements in the *water meter* use case and discuss the design and meticulous mitigation provided by TEE-Watchdog complementing the TrustZone security features and eliminating the risk of the abovementioned threats.

Operational/Functional Requirements: the components of a *water meter*' s operational environment include (i) O1: a flow sensor, which measures water consumption, (ii) O2: LPWAN transceiver (TRX), which provides low-power communication, (iii) O3: additional *water meter* sensors such as the battery sensor, and pH, temperature, conductivity, or any other water quality sensors, and (iv) O4: *water meter* display, which shows readings to the user. A networked *water meter*'s operational environment includes (i) O5: back-end servers, used for uploading measurements and administrating the meters, (ii) and O6: gateways to aggregate communications between devices and back-end server.

Security Requirements: the security requirements include (i) S1: the integrity of the *water meter* ID; (ii) S2: the integrity and authenticity of the secure firmware; (iii) S3: integrity and confidentiality of the flow sensor measurements; and (iv) S4: confidentiality sensor data (battery sensor, water quality sensors, etc.) in the operational environment.

Designing a *Water Meter*: we present the *water meter* system design (Figure 8) fulfilling the functional and security requirements. The *water meter* is designed on a Cortex-M23/M33 microcontroller with TrustZone-M security extensions enabled to support a TEE. The device includes an embedded flash memory to store the secure firmware, firmware certificate, manifest file for each sensor, and manifest file certificates (S2). A one-time-programmable (OTP) or tamper-resistant trusted storage is included to store *water meter* ID, HUKs, log file, etc. (S1). An LPWAN controller is included and mapped in the normal world to minimize the code base of the secure domain and minimize attack surface (O2). A flow sensor is integrated onto the microcontroller and mapped to the secure domain (O1). The firmware of the IoT device is composed of a secure bootloader, which is the first piece of code called by the ROM; a minimal secure kernel, which manages all the critical operations, mode switches, and TCBs and loads the normal domain's OS. The normal OS supports the application software running on top of this OS. Measuring water flows is the role of *water meter* software. Flow measurements are recorded and fed as input to the water distributor's billing system. The *water meter* software is designed in two main modules: the secure module processes the sensor data to convert it into volume or other analytically useful forms, and the normal world module receives the aggregated sensor values in encrypted form (S3) from the secure domain and transmits the measurements to a local gateway or a central server (O5, O6). The normal world module of the software is also responsible for displaying the reading for the user on a connected display (O4). The device manufacturer integrates water quality and battery sensors on the device as well (O3). TEE-Watchdog protection mechanisms ensure that the *water meter*'s software does not access data from other sensors located in the secure domain with an intent to transmit outside the device (S4).

Implementing a *Water Meter* with TEE-Watchdog Protections: TEE-Watchdog components are introduced in the firmware and as part of the secure kernel that manages TEE. Approximately 460 lines of code were added to implement TEE-Watchdog. The bootup process of the *water meter* device is delayed by 1450 microseconds due to manifest file verification and translating the 4 policies (see the following listing). All those components whose policies are not defined are by default not accessible to the software.

```
{  "UniqueID"  : "AD-4E-22-C5-61-FF-AF",
   "Policies"  : {
                 "Flow-sensor"           : "RW",
                 "pH-sensor"             : "NA",
                 "Temperature-sensor"    : "RO",
                 "Conductivity-sensor"   : "NA"
                 }
}
```

The CBOR encoding of the *water meter*'s manifest file is listed below.

```
A2                                              # map(2)
   68                                           # text(8)
      556E697175654944                          # "UniqueID"
   74                                           # text(20)
      41442D34452D32322D43352D36312D46462D4146  # "AD-4E-22-C5-61-FF-AF"
   68                                           # text(8)
      506F6C6963696573                          # "Policies"
   A4                                           # map(4)
      6B                                        # text(11)
         466C6F772D73656E736F72                 # "Flow-sensor"
      62                                        # text(2)
         5257                                   # "RW"
      69                                        # text(9)
         70482D73656E736F72                     # "pH-sensor"
      62                                        # text(2)
         4E41                                   # "NA"
      72                                        # text(18)
         54656D70657261747572652D73656E736F72   # "Temperature-sensor"
      62                                        # text(2)
         524F                                   # "RO"
      73                                        # text(19)
         436F6E6475637469766974792D73656E736F72 # "Conductivity-sensor"
      62                                        # text(2)
         4E41                                   # "NA"
```

The *water meter* application is divided into two modules, the secure and the nonsecure. The normal/nonsecure part of the application resides in the normal world and deals with operations related to transferring the encrypted sensor readings to a server for further processing. The secure part of the application residing in the secure world obtains the water

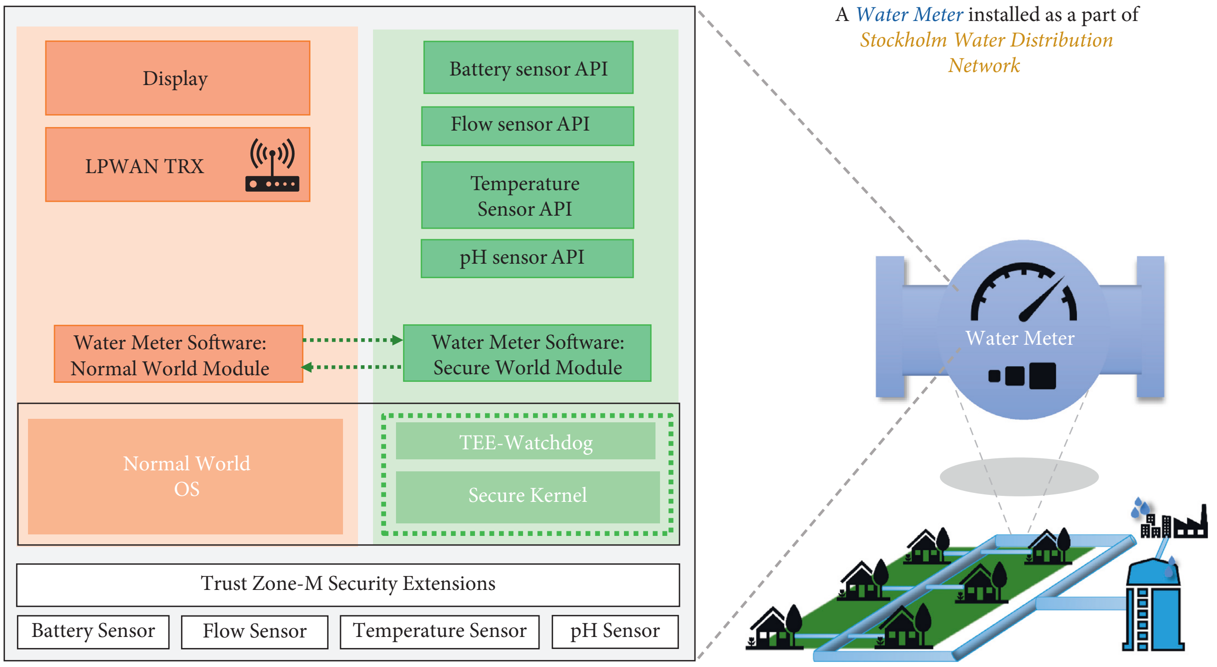

Display
LPWAN TRX
Battery sensor API
Flow sensor API
Temperature Sensor API
pH sensor API
Water Meter Software: Normal World Module
Water Meter Software: Secure World Module
Normal World OS
TEE-Watchdog
Secure Kernel
Trust Zone-M Security Extensions
Battery Sensor
Flow Sensor
Temperature Sensor
pH Sensor
A Water Meter installed as a part of Stockholm Water Distribution Network
Water Meter

Figure 8: Components of a *water meter* in Stockholm water distribution network supporting TrustZone-M and TEE-Watchdog protections.

flow sensor reading using the flow sensor drivers and requests crypto-services to encrypt the reading so it can be transferred over the network.

The primary function of the *water meter* application is reading from a secure water flow sensor, converting raw data value into a useful reading, and transmitting the reading. In this section, we discuss the overhead of TEE-Watchdog on the execution of *water meter* application: (i) runtime overhead on memory (RAM), (ii) runtime delay in CPU time due to enabling protections, and (iii) end-to-end latency. End-to-end latency here refers to one complete computational cycle starting from a request to the secure water flow sensor by the *water meter* application from the normal world and ending when the control is returned back to the normal world. TEE-Watchdog consumes 5.47% of the total available RAM. On every context switch to the secure domain, TEE-Watchdog protects the sensors according to the access table. These protections come at a cost of 208 microseconds CPU time delay in execution. This slight delay during context switch affects the end-to-end latency of sensor access by 156 microseconds, which is 5.6%. Figure 7(b) shows the impact of TEE-Watchdog on the latency of applications in detail. In this use case, a delay of 156 microseconds does not significantly affect any real-time operation such as billing or water usage. The most time and energy-consuming operation of the TEE-Watchdog protection lifecycle is registering an illegal behaviour to the log file. The integrity and confidentiality of the log file are critical to the security of the device, and a trusted secure storage (TrustZone-M's SST) is used to store the log file. TrustZone-M's SST is hardware-protected storage where data are encrypted and stored. Writing to SST takes 0.76 milliseconds per one byte of data written, and hence, one log file entry takes 50.16 milliseconds.

11. Security Analysis

We have introduced TEE-Watchdog, our proposed framework to restrict access to secure peripherals and efficiently generate records of misbehaving software. We use a number of mechanisms to design TEE-Watchdog such as MPU-based protections, fault handling, and CBOR decoding and parsing. In this section, we provide a security evaluation of our proposed solution and its modules. We consider an adversary A1, A2, and A3 as discussed in our threat model (Section 3) whose main goal is to bypass TEE-Watchdog protections and gain access to peripheral data beyond its permissions. The ultimate threat here is data exfiltration from an IoT device.

We revisit each of the security goals of TEE-Watchdog: **G1**, **G2,** and **G3** enlisted in our threat model and discuss the potential attack surfaces of TEE-Watchdog modules and how they are mitigated through our design choices of TEE-Watchdog.G1: secure applications cannot modify TEE-Watchdog components and structures.

(i) The IoT device manufacturer (acting as class A2 or A3 attacker as defined in Section 3) can ship the devices to the IoT solution provider with unwanted changes done in the TEE-Watchdog itself. Our design approach follows that the IoT solution provider, who is considered trusted, should be able to check the integrity of the TEE-Watchdog before the deployment of IoT devices. The proposed TEE-Watchdog implements a security manager, which is privileged software within a standard OS running in the secure world. It is therefore possible to verify the integrity of the TEE-Watchdog by computing and comparing the hash of the TEE-Watchdog binary with the published good hash of the TEE-Watchdog just like OS integrity is normally verified. Hence, any malicious attempt to change the TEE-Watchdog can be easily detected by the IoT solution provider.

(ii) Access table can be modified by any secure software in TEE to change access permissions of itself or another process. TEE-Watchdog is designed such that the access table is created by Security Manager, which is privileged secure software. This makes the access table a part of the privileged memory region and becomes inaccessible to other access levels.

(iii) Log file entries can be accessed and read by secure software since it is a secure resource. To guarantee the confidentiality of the log file, it is placed encrypted in secure storage (TFM-SST) against the application ID of the Security Manager.

Log file entries can be falsified by secure software either by duplicating the existing entries or creating new false entries. TEE-Watchdog leverages the TF-M client ID management system and task control block (TCB) in the secure world to associate log file entries to the software making the entry. This easily helps to identify whether the log file entry was created by any other software packages besides the Security Manager.

(i) The log file entries can be overwritten by malicious software by repeatedly creating access faults and depleting the capacity of the log file. As mitigation, before making a new entry, the Security Manager checks the number of existing entries in the log file against the maxEntries; it sends the file for further processing if the existing entries are equal to the capacity of the log file.

(ii) Manifest file can be manipulated by adding falsified policies before it is even translated to access table. The authenticity of the manifest file can be validated beforehand by using conventional certificates or signature management schemes decided between vendors.

G2: normal world applications and their trusted code cannot interrupt TEE-Watchdog operations and processes that make TEE-Watchdog functional.

(i) The manifest file parsing process can be interrupted by secure software. TEE-Watchdog is designed to perform parsing of the encoded manifest file once when the system boots up as part of the secure boot process, it does not allow runtime parsing of new policies, hence mitigating any chance of interruption of this process.

G3: malicious applications are prevented from depleting TEE-Watchdog resources.

(i) Log file entries can deplete the limited storage space available on resource-constrained devices. After a predefined number of limiting entries in the log file, it is programmed to be uploaded to an external entity for further processing or storage. This eliminates the risk of accidental or malicious overwriting of the log file.

12. Conclusion

We have presented TEE-Watchdog, to bring trust within an IoT device composed of heterogeneous components supplied by multiple vendors. TEE-Watchdog is a mechanism to restrict software access to secure system peripherals based on predefined security policies and permissions. TEE-Watchdog introduces a compact CBOR-encoded manifest file template for device vendors/manufacturers to use for specifying access policies. TEE-Watchdog also enables efficient behavioral logging of misbehaving software. TEE-Watchdog leverages the ARM MPU to create memory restrictions, uses the fault handling mechanism to log misbehaviour, and utilizes standard CBOR encoding and decoding mechanism to parse the compact CBOR-encoded manifest file. We have implemented TEE-Watchdog in a TrustZone-M-enabled IoT device and evaluated its execution overhead and performance. Our microbenchmark evaluation of the proof-of-concept implementation shows that additional operations introduced with TEE-Watchdog are at par with normal system operations. There is a 1.4% delay in latency of peripheral access due to TEE-Watchdog protections. We also highlight the advantage of using our proposed CBOR-encoded template for the policies as compared to the standard JSON file format. The 40% reduction in size of the manifest file is a marginal gain considering the constrained nature of the IoT devices. We also show the applicability of TEE-Watchdog framework through a water meter use case by ARM.

Data Availability

We have no associated data sets used in the study. The source code used to support the findings of this study can be made available to the reviewers from the corresponding author upon request for the review process. The authors, however, do not plan to open source the code just yet as they will be using it within projects in the projects' remaining timeline.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was partly supported by the Swedish Foundation for Strategic Research (SSF) aSSIsT and RISE KP and partly by the EU H2020 VEDLIoT (GA No. 957197) and EU H2020 CONCORDIA (GA No. 830927) projects.

References

[1] Samsung, "Samsung smartthings," 2020, https://www.smartthings.com/.

[2] Apple, “Apple homekit,” 2020, https://developer.apple.com/homekit/.

[3] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, “SoK: understanding the prevailing security vulnerabilities in trustzone-assisted TEE systems,” in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pp. 18–20, San Francisco, CA, USA, May 2020.

[4] S. Mirzamohammadi, J. A. Chen, A. A. Sani, S. Mehrotra, and G. Tsudik, “Ditio: trustworthy auditing of sensor activities in mobile & iot devices,” in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, November 2017.

[5] Arm Ltd, “ARM security technology, building a secure system using TrustZone technology,” 2019, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html.

[6] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, “Innovative technology for cpu based attestation and sealing,” in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13, ACM, Tel-Aviv Israel, June 2013.

[7] D. Lee, D. Kohlbrenner, S. Shinde, D. Song, and K. Asanović, “Keystone: a framework for architecting tees,” 2019, https://arxiv.org/abs/1907.10119.

[8] X. Ge, H. Vijayakumar, and T. Jaeger, “Sprobes: enforcing kernel code integrity on the trustzone architecture,” 2014, https://arxiv.org/abs/1410.7747.

[9] Z. Ning, F. Zhang, W. Shi, and W. Shi, “Position paper: challenges towards securing hardware-assisted execution environments,” in *Proceedings of the Hardware and Architectural Support for Security and Privacy*, pp. 1–8, Toronto ON Canada, June 2017.

[10] Arm Ltd, “TrustZone technology for the ARMv8-M architecture,” 2019, https://static.docs.arm.com/100690/0200/armv8m_trustzone_technology_100690_0200.pdf.

[11] R. Strackx, P. Frank, and B. Preneel, “Efficient isolation of trusted subsystems in embedded systems,” in *Proceedings of the International Conference on Security and Privacy in Communication Systems*, pp. 344–361, Springer, Singapore, September 2010.

[12] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, “Tytan: tiny trust anchor for tiny devices,” in *Proceedings of the 52nd annual design automation conference*, pp. 1–6, San Francisco, CA, USA, June 2015.

[13] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, “Trustlite: a security architecture for tiny embedded devices,” in *Proceedings of the Ninth European Conference on Computer Systems*, April 2014.

[14] I. Anati, S. Gueron, S. Johnson, and S. Vincent, “Innovative technology for cpu based attestation and sealing,” in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13, p. 7, Citeseer, Tel-Aviv Israel, June 2013.

[15] H. Chung, T. Kim, H. Choi et al., “Securing real-time microcontroller systems through customized memory view switching,” in *Proceedings of the Network and Distributed System Security Symposium NDSS*, San Diego, California, February 2018.

[16] A. A. Clements, Naif Saleh Almakhdhub, S. Bagchi, and M. Payer, “Automatic compartments for embedded systems,” in *Proceedings of the 27th USENIX Security Symposium USENIX*, pp. 65–82, BALTIMORE, MD, USA, August 2018.

[17] A. Machiry, E. Gustafson, C. Spensky et al., “Boomerang: Exploiting the semantic gap in trusted execution environments,” in *Proceedings of the Network and Distributed System Security Symposium NDSS*, San Diego, California, March 2017.

[18] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, “Armageddon: cache attacks on mobile devices,” in *Proceedings of the 25th USENIX Security Symposium USENIX*, pp. 549–564, Austin TX USA, August 2016.

[19] D. Rosenberg, *Reflections on Trusting Trustzone*, BlackHat, 2014.

[20] I. Skochinsky, *Intel Me Secrets.* Code Blue, 2014.

[21] S. Embleton, S. Sparks, and C. C Zou, “Smm rootkit: a new breed of os independent malware,” *Security and Communication Networks*, vol. 6, no. 12, pp. 1590–1605, 2013.

[22] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, “Flowfence: practical data protection for emerging iot application frameworks,” in *Proceedings of the 25th USENIX Security Symposium USENIX*, pp. 531–548, Austin, TX, USA, August 2016.

[23] S. Holavanalli, D. Manuel, V. Nanjundaswamy et al., “Flow permissions for android,” in *Proceedings of the 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 652–657, IEEE, Silicon Valley, CA, USA, November 2013.

[24] T. Denning, T. Kohno, and H. M. Levy, “Computer security and the modern home,” *Communications of the ACM*, vol. 56, no. 1, pp. 94–103, 2013.

[25] E. Fernandes, J. Jung, and A. Prakash, “Security analysis of emerging smart home applications,” in *Proceedings of the 2016 IEEE symposium on security and privacy (SP)*, pp. 636–654, IEEE, San Jose, CA, USA, May 2016.

[26] Lu. Zhao, G. Li, B. De Sutter, and J. Regehr, “Armor: fully verified software fault isolation,” in *Proceedings of the ninth ACM international conference on Embedded software*, pp. 289–298, Taipei, Taiwan, October 2011.

[27] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, “Armlock: hardware-based fault isolation for arm,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 558–569, Scottsdale Arizona USA, November 2014.

[28] Zelalem Birhanu Aweke and A. Todd, “Usfi: ultra-lightweight software fault isolation for iot-class devices,” in *Proceedings of the 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1015–1020, IEEE, Dresden, Germany, March 2018.

[29] L. Batina, P. Jauernig, N. Mentens, A.-R Sadeghi, and E. Stapf, “Hardware we trust: gains and pains of hardware-assisted security,” in *Proceedings of the 56th Annual Design Automation Conference 2019*, Las Vegas NV USA, June 2019.

[30] Arm Ltd, “ARMv8-M memory protection unit,” 2017, https://static.docs.arm.com/100699/0200/armv8m_memory_protection_unit_100699_0200_en.pdf.

[31] Android, “App manifest overview,” 2021, https://developer.android.com/guide/topics/manifest/manifest-intro.

[32] “Manifest,” *Ubuntu*, https://wiki.ubuntu.com/SecurityTeam/Specifications/ApplicationConfinement/Manifest#Use_in_Ubuntu, 2021.

[33] P. H. C. Bormann, *Concise binary object representation (cbor)*, p. 10, RFC Editor, 2013.

[34] Bo. Petersen, H. Bindner, Y. Shi, and B. Poulsen, “Smart grid serialization comparison: comparision of serialization for distributed control in the context of the internet of things,” in *Proceedings of the 2017 Computing Conference*, pp. 1339–1346, IEEE, London, UK, July 2017.

[35] Aioti Wg03-loT Standardisation, *High Level Architecture (Hla)*, Technical specification, 2017.

[36] Information technology — structure for the identification of organizations and organization, "Parts — part 1: identification of organization identification schemes. Standard," International Organization for Standardization, Geneva, CH, 1998.

[37] Ieee Standards Association, "Guidelines for use of extended unique identifier (eui), organizationally unique identifier (oui), and company id (cid)," 2017, https://standards.ieee.org/content/dam/ieee-standards/standards/web/documents/tutorials/eui.pdf.

[38] F. Sadique, C. Sui, I. Vakilinia, S. Badsha, and S. Sengupta, "Automated structured threat information expression (stix) document generation with privacy preservation," in *Proceedings of the 2018 9th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pp. 847–853, IEEE, New York, NY, USA, November 2018, https://doi.org/10.1109/UEMCON.2018.8796822.