TZ-DataShield: Automated Data Protection for Embedded Systems via Data-Flow-Based Compartmentalization

Zelun Kong* Minkyung Park* Le Guan† Ning Zhang‡ Chung Hwan Kim*
*University of Texas at Dallas †University of Georgia ‡Washington University in St. Louis
*{zelun.kong, minkyung.park, chungkim}@utdallas.edu †leguan@uga.edu ‡zhang.ning@wustl.edu

***Abstract*—As reliance on embedded systems grows in critical domains such as healthcare, industrial automation, and unmanned vehicles, securing the data on micro-controller units (MCUs) becomes increasingly crucial. These systems face significant challenges related to computational power and energy constraints, complicating efforts to maintain the confidentiality and integrity of sensitive data. Previous methods have utilized compartmentalization techniques to protect this sensitive data, yet they remain vulnerable to breaches by strong adversaries exploiting privileged software.**

In this paper, we introduce TZ-DataShield, a novel LLVM compiler tool that enhances ARM TrustZone with *sensitive data flow (SDF)* compartmentalization, offering robust protection against strong adversaries in MCU-based systems. We address three primary challenges: the limitations of existing compartment units, inadequate isolation within the Trusted Execution Environment (TEE), and the exposure of shared data to potential attacks. TZ-DataShield addresses these challenges by implementing a fine-grained compartmentalization approach that focuses on sensitive data flow, ensuring data confidentiality and integrity, and developing a novel intra-TEE isolation mechanism that validates compartment access to TEE resources at runtime. Our prototype enables firmware developers to annotate source code to generate TrustZone-ready firmware images automatically. Our evaluation using real-world MCU applications demonstrates that TZ-DataShield achieves up to 80.8% compartment memory and 88.6% ROP gadget reductions within the TEE address space. It incurs an average runtime overhead of 14.7% with CFI and DFI enforcement, and 7.6% without these measures.

I. Introduction

The emergence of the Internet of Things and cyber-physical systems has significantly increased our reliance on embedded systems across critical domains such as healthcare, industrial automation, and unmanned vehicles. Many of these devices utilize microcontroller units (MCUs) for their low cost, compact size, and energy efficiency. However, these constraints often lead firmware developers to sacrifice security. For instance, to reduce costs and avoid the runtime overhead associated with memory isolation, many MCU-based systems operate with a shared address space. This allows any component within the system to directly access all computing resources, including code, data, and peripherals, as they are mapped into the shared address space without any security mechanisms in place.

This practice poses significant risks, particularly when there is a need to protect the confidentiality and integrity of data, as many MCU-based systems store sensitive information in the shared address space. For instance, a PIN stored in a door lock system or GPS coordinates in an unmanned vehicle system requires protection from malicious eavesdropping and manipulation attacks [1], [2], [3], [4], [5], [6], [7], [8]. Adversaries may reveal or tamper with other memory regions containing full or partial information on sensitive data (*e.g.*, a variable that holds a GPS coordinate) and exploit any code that accesses those memory regions.

Existing works have proposed various compartmentalization techniques to minimize the attack surface within the shared address space of MCU-based systems [9], [10], [11], [12]. These techniques divide firmware into multiple compartments with varying levels of granularity, such as threads [9], [10], [11], software components [10], [11] (*e.g.*, kernel, device drivers), and functions [12]. A security monitor then isolates memory accesses at runtime using the Memory Protection Unit (MPU) [9], [10], [12] or language-based code instrumentation [11], ensuring each compartment can only access resources within its protection domain.

Unfortunately, the limited number of protection domains (typically eight) presents a challenge, as many MCU applications require more than eight regions to isolate each compartment. Previous works [12], [9], [10], [11] have attempted to mitigate this limitation by merging multiple compartments into one. However, this approach increases the size and complexity of the isolated memory region, thereby creating new attack surfaces. To improve the security of each compartment, it is crucial to isolate them individually with adequate granularity.

Additionally, although these compartmentalization techniques reduce the likelihood of malicious access to sensitive data, they operate under the assumption that privileged software (*e.g.*, the RTOS kernel) within the Trusted Computing Base (TCB) remains secure. However, with the increasing complexity and number of vulnerabilities in embedded OSes (*e.g.*, CVE-2021-43997 in FreeRTOS), the barrier to compromising the kernel has been significantly lowered [13], [14], [15], [16].

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240563
www.ndss-symposium.org

In this paper, we leverage both ARM TrustZone and compartmentalization to protect sensitive data in MCU-based systems against strong adversaries operating in privileged mode. Specifically, our data flow analysis traces all data flows related to the sensitive data to be protected. Using TrustZone, a system can allocate sensitive resources to a trusted execution environment (TEE), known as *the secure world*, and isolate them from any component running in the untrusted execution environment, known as *the normal world*, regardless of the privilege mode of the component in the normal world. This isolation ensures that the security monitor is protected from such attacks. However, simply adopting TrustZone introduces new challenges that must be addressed to build an effective data protection mechanism for MCU-based systems:

Challenge 1: Compartment Granularity. Existing compartmentalization techniques are either too coarse-grained (*e.g.*, thread), resulting in a wider attack surface, or unnecessarily fine-grained (*e.g.*, function), leading to high compartment switch overhead. We propose a compartmentalization technique based on *sensitive data flow (SDF)*, specifically designed to protect data confidentiality and integrity in MCU-based systems. This compartment unit reduces the attack surface by including only the minimal computing base necessary.

Challenge 2: Lack of Intra-TEE Isolation. Using TrustZone, the memory regions that contain sensitive data and the code that accesses them can be allocated to the secure world, isolating them from potential adversaries in the normal world. Still, it does not provide any isolation within the secure world itself. Without intra-TEE isolation, adversaries could compromise vulnerable code running in the secure world, access sensitive data, and potentially cause a full system compromise. To address this, we leverage *software fault isolation (SFI)* to isolate unverified computations within each compartment, thereby mitigating this problem. We opt for a software solution to avoid the MPU's hardware limitations on the number, size, and alignment of protected regions.

Challenge 3: Shared and External Resources. Similar to existing compartmentalization techniques, our SDF compartments may share data with other compartments (*e.g.*, a global variable) or legitimately access peripherals in the secure-world address space. Adversaries may exploit this to illegally access another compartment's sensitive data from a compromised compartment or maliciously control a peripheral allocated to the compartment. To address this, we develop a lightweight mechanism to enforce *Control Flow Integrity (CFI)* and *Data Flow Integrity (DFI)*, ensuring that these shared resources are accessed only through verified control and data flows.

In light of these challenges, we developed a new LLVM compiler tool, TZ-DATASHIELD, which firmware developers can easily use to compartmentalize an MCU-based system and protect sensitive data by leveraging TrustZone. With the C/C++ preprocessors provided by TZ-DATASHIELD, developers can annotate the source code to mark each sensitive data item (*e.g.*, global, stack, heap variable, and peripheral-mapped region) with a macro indicating read, write, or read+write permissions. During compilation, the firmware is automatically compartmentalized to produce **(1)** the *secure-world firmware*, **(2)** one or more *SDF compartments*, which contain a piece of code and necessary metadata, and **(3)** the *normal-world firmware*. The secure world firmware contains a trusted *security monitor* that securely loads each compartment from the normal world into the secure world. The SDF compartments are identified using forward and backward program slicing, with the sensitive data serving as the source and sink, respectively. Consequently, each compartment contains only the code and data necessary for accessing a specific piece of sensitive data. The normal-world firmware contains the rest of the system that does not need to access any sensitive data.

To enforce intra-TEE isolation, TZ-DATASHIELD instruments every branch and memory access instruction within each compartment, confining potential vulnerabilities and attacks to that compartment. This prevents illegal access from one compartment to other resources in the secure world, including those of other compartments and the security monitor. Furthermore, since a compartment may *legitimately* share data with another compartment or access external secure-world resources outside, which SFI should not restrict, we developed a lightweight CFI/DFI mechanism. This mechanism ensures that access to these resources is only permitted after successful verification of the control and data flow.

We implemented a prototype of TZ-DATASHIELD and evaluated it on the LPCXpresso55S69 [17] development board with an ARMv8-M-based MCU. In our security evaluation with various bare-metal and RTOS firmware, TZ-DATASHIELD achieved up to 80.8% compartment memory and 88.6% ROP gadget reductions in the secure-world address space while protecting various sensitive data. Our evaluation also demonstrated that TZ-DATASHIELD can eliminate different types of attacks in the RTOS and effectively block attacks from a strong adversary targeting sensitive data in the secure world. The average runtime overhead of TZ-DATASHIELD is 14.7% with both CFI and DFI enabled and 7.6% without them, demonstrating a 36.7% speedup compared to function-based compartmentalization. TZ-DATASHIELD requires less than 45 KB of flash memory and 7 KB of RAM to provide this protection, which is comparable to the overhead of coarse-grained compartmentalization (component and thread) and 30

In summary, the contributions of this paper are as follows:

- We propose TZ-DATASHIELD, a novel LLVM compiler tool that automatically compartmentalizes MCU firmware based on sensitive data flow. The compartments are executed in the secure world and protected by TrustZone, ensuring the confidentiality and integrity of sensitive data against strong adversaries in the normal world.
- We design a data-flow-based compartmentalization technique called SDF compartmentalization, which achieves sensitive data flow protection and intra-TEE isolation. It also enables the protection of shared and peripheral data. Additionally, TZ-DATASHIELD reduces the attack surface and TCB size of the secure world firmware.
- We implement and open-source a prototype of TZ-DATASHIELD [18] with an easy-to-use annotation sys-

tem for marking sensitive data to be protected. TZ-DATASHIELD produces TrustZone-ready firmware images that securely execute SDF compartments in isolation from potential attacks in the normal world.
- We evaluate our prototype on various bare-metal and RTOS firmware and conduct comprehensive security and performance evaluations. Our results demonstrate that TZ-DATASHIELD can protect sensitive data from strong adversaries exploiting vulnerabilities in the compartments, achieving up to 80.8% reduction in the secure-world memory compared with other compartmentalization techniques.

II. BACKGROUND OF ARMV8-M TRUSTZONE

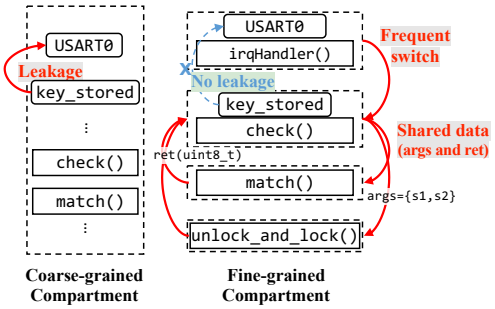
Privilege Modes and ARM TrustZone. ARM processors for MCUs support two privilege modes to control access to system resources: privileged and unprivileged modes. The privileged mode is typically reserved for system software, such as the RTOS kernel, and the processor prevents software in the unprivileged mode from directly accessing the privileged mode software. Many security mechanisms for MCUs rely on the privileged mode to protect their security monitor from malicious attacks [9], [10], [12], [19], [20], [11]. However, there have been advanced attacks on MCU-based systems, capable of compromising software in the privileged mode, *e.g.*, by exploiting a vulnerability in an RTOS [16], [13], [14], [15]. Such attacks can disable existing security mechanisms that rely on the privileged mode and lead to a full system compromise.

To counteract such strong adversaries, the ARMv8-M architecture introduces TrustZone for MCUs. TrustZone segregates the processor into two distinct environments, called the *secure world* and the *normal world*, facilitating strong and hardware-based isolation between the two. The secure world provides a TEE where sensitive computing resources can be allocated, while the normal world is for general RTOS and applications, which are assumed to be potentially compromised. The hardware ensures that any software running in the normal world is isolated from the secure world, regardless of the privilege modes. As such, even privileged software in the normal world (*e.g.*, compromised or malicious RTOS) cannot access any resource in the secure world.

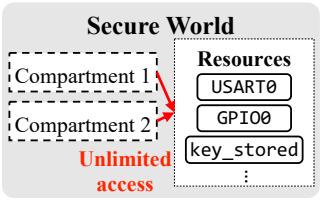
Security Attribution Unit (SAU). The security of resources in ARMv8-M TrustZone is controlled by a hardware component, the SAU. Similar to an MPU, the entire address space can be partitioned into a specified number of memory regions, called the *SAU regions*. Each SAU region is configured with a security attribute, *e.g.*, to allocate it to the secure world or the normal world. This mechanism extends to allocating peripherals to the secure or normal world as peripherals are also mapped to the address space, similar to the flash memory and SRAM. Notably, the SAU allows dynamic updates to region configurations and their access policies. It allows TZ-DATASHIELD to reallocate compartments between the secure world and the normal world at runtime, keeping unnecessary compartments outside the secure world.

```
1   /* Data from key_stored is confidential */
2   const uint8_t key_stored[SIZE] = {0x1234567890abcdef};
3   uint8_t key_received[SIZE] = {0};
4   int main(void) {
5     // Peripheral initialization ...
6     for (;;) {
7       sleep(interval);
8       check();
9     }
10  }
11  void check(void) {
12    if (match(key_stored, key_received))
13    unlock_and_lock();
14  }
15  bool match(uint8_t *s1, uint8_t *s2) {
16    for (uint32_t i = 0; i < SIZE; i++)
17      if (s1[i] != s2[i])  return false;
18    return true;
19  }
20  void unlock_and_lock(void) {
21    PinWrite(LOCK_PORT, LOCK_PIN, 1);
22    PinWrite(LOCK_PORT, LOCK_PIN, 0);
23  }
24  void irq_handler(void) {
25    /* Data from USART must not be manipulated */
26    usart_data msg = DEVICE_READ(USART);
27    switch(msg.type) {
28      case KEY_CHAR:
29        key_received[rx_index] = msg.data;
30      // Other cases ...
31    }
32  }
```

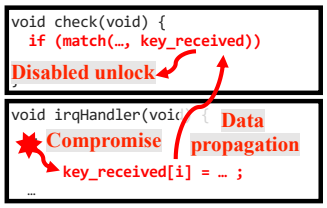
Listing 1: A code snippet of a smart door lock system, PinLock. The code is simplified for readability.



(a) Inadequate compartment granularity.



(b) Lack of intra-TEE isolation.



(c) Compromised shared data.

Figure 1: Challenges in protecting sensitive data of PinLock. The individual data flows of `key_received` and `key_stored` must be isolated considering various attack surfaces.

III. MOTIVATING CHALLENGES

In this section, we outline three key challenges that we have identified while protecting sensitive data in MCU-based systems. Listing 1 shows a code snippet of the PinLock system as an example. An interrupt handler `irqHandler()` receives a

PIN entered by the user from a keypad by reading a peripheral-mapped region at a certain address `USART` and storing it in `key_received` (lines 29-38). If the received PIN corresponds with the stored key in `key_stored` (lines 13-22), the door unlocks by writing to the lock peripheral (lines 24-27). This authentication method works only when the data flows of the sensitive data in `key_stored` remains confidential and the other data from the keypad at `USART` has not been tampered with. Figure 1 illustrates the challenges that TZ-DATASHIELD aims to address to protect the confidentiality and integrity of these sensitive data.

SDF Compartmentalization. Table I presents different compartmentalization techniques for MCU firmware in comparison. Although various compartmentalization approaches have been proposed for MCUs [9], [10], [12], [11], the compartment units (*i.e.*, granularities) of these approaches are not suitable for protecting sensitive data flow. In particular, Minion [9], EC [10], and CRT-C [11] employ coarse-grained compartment units such as threads, components (*e.g.*, libraries or RTOS kernel), and devices. While ACES [12] provides fine-grained compartmentalization of distinct functions. These compartment units, however, fall short in defending against attacks that exploit code vulnerabilities to breach confidentiality and integrity. This is because a single sensitive data flow might influence others within the same compartment.

As shown in Figure 1(a), although coarse-grained compartmentalization restricts access to resources for each thread, component, or device, it may include more resources than necessary in the compartment to handle a single piece of sensitive data in the compartment, resulting in the potential leakage of other sensitive data. For example, a thread-based compartment will include the data flows of both `key_stored` and `USART` in one compartment because they belong to the same thread in the system, and an attack that compromises the execution of `key_stored` will also be able to leak the received PIN from the keypad at `USART`.

On the other hand, fine-grained compartmentalization reduces the attack surface by dividing the resources into more small compartments (*e.g.*, one compartment per function). However, it suffers from a high runtime overhead caused by more frequent switching between the compartments (*e.g.*, function calls). In addition, it forces multiple compartments to have more shared data than coarse-grained compartments across the compartment boundaries, which attackers may target, resulting in a limited attack surface reduction, such as function argument and return data. Yet, this approach may incur significant overhead due to the necessity for frequent compartment switching from function calls and returns as illustrated in Figure 1(a).

To address this problem, our approach focuses on compartmentalizing distinct *sensitive data flows* to protect the confidentiality and integrity of sensitive data. By grouping all relevant functions into one compartment, *SDF compartmentalization* solution not only offers the expected isolation but eliminates unnecessary compartment switching, thereby addressing the limitations in the previous compartmentalization efforts. In the PinLock example, our approach makes one compartment for the read accesses to key_stored to ensure its confidentiality and the other for the write access to `key_received` to its integrity. Functions such as `unlock_and_lock()`, `check()`, and `match()` would belong to one compartment (say, Compartment 1) for `key_stored`. Meanwhile, for the integrity of `key_received`, the function `irqHandler()` would constitute another compartment (say, Compartment 2).

Intra-TEE Isolation for TrustZone. Prior compartmentalization approaches are based on a weak attack model assuming that the adversary cannot compromise the privileged software. Under this attack model, their compartment isolation mechanisms run in the same privilege mode with other privileged software (*e.g.*, RTOS kernel) and leverage an MPU or memory access checks inserted by a memory-safe language to enforce compartment boundaries at runtime. However, a strong adversary taking control of the privileged software (*e.g.*, by exploiting its vulnerabilities such as CVE-2021-43997) can bypass these mechanisms (§II). Hence, our strategy involves segregating all compartments reliant on sensitive data into the secure world, with other computations occurring in the normal world using TrustZone.

Unfortunately, TrustZone does not provide an isolation mechanism within the secure world [15], [14], [21]. Consequently, all programs within the secure world are trusted, including unverified code in compartments with potential vulnerabilities, allowing them access to any resources allocated to the secure world. This weakness could enable an adversary to exploit a vulnerability within a compartment, thereby gaining unlimited access to sensitive data across compartments in the absence of intra-TEE isolation measures. For instance, compromised code within `irqHandler()` could potentially retrieve and leak `key_stored` (Figure 1(b)).

To mitigate such threats, our approach leverages SFI to ensure that operations, like reading `key_stored` in Compartment 1, are not abused to access other secure-world data (*e.g.*, the keypad at `USART`) or code exclusively assigned to Compartment 2. Although the MPU can be used to design an intra-TEE isolation mechanism for TrustZone, we avoid using it due to its inherent limitations [13]: (1) limited number of protected memory regions (typically eight) (2) constraints on memory region size and alignment. For the second reason, a compartment usually consumes at least two regions – one for data and another for code – as they are not contiguous in memory space. This further exacerbates the limitation regarding the number of memory regions. Previous works [12], [9], [10], [11] have circumvented this limitation by merging multiple compartments into one. However, this approach increases the size of the isolated memory region, thereby creating new attack surfaces. Therefore, to ensure the security of each compartment, it is necessary to isolate them individually with adequate granularity.

Although there exists research to develop intra-TEE isolation techniques for ARM TrustZone, their design depends on special hardware features that are not available on ARM MCUs and focuses on isolating certain software components

Table I: Comparison of different compartmentalization techniques for MCU-based systems.

Project	Compartment Units	Compartment Isolation	Compartment Switch	Attack Model	Shared Data Protection	# Compartments	Target Firmware
Minion [9]	Thread	MPU + SVC	Infrequent	Weak†	None	Limited	RTOS
EC [10]	Thread, Component, Device	MPU + DWT	Infrequent	Weak†	Type Extensions	Limited	Bare-metal, RTOS
ACES [12]	Function(s)	MPU + SVC	Frequent	Weak†	Dynamic Profiling	Limited	Bare-metal
CRT-C [11]	Thread, Component	Language-based Isolation	Infrequent	Weak†	Object Sharing	No Limits	RTOS
TZ-DATASHIELD	**Sensitive Data Flow**	**TrustZone + Intra-TEE Isolation**	**Moderate**	**Strong**	**CFI + DFI**	**No Limits**	**Bare-metal, RTOS**

† An attacker who compromises software in the privileged execution mode (*e.g.*, RTOS) can bypass the protection.

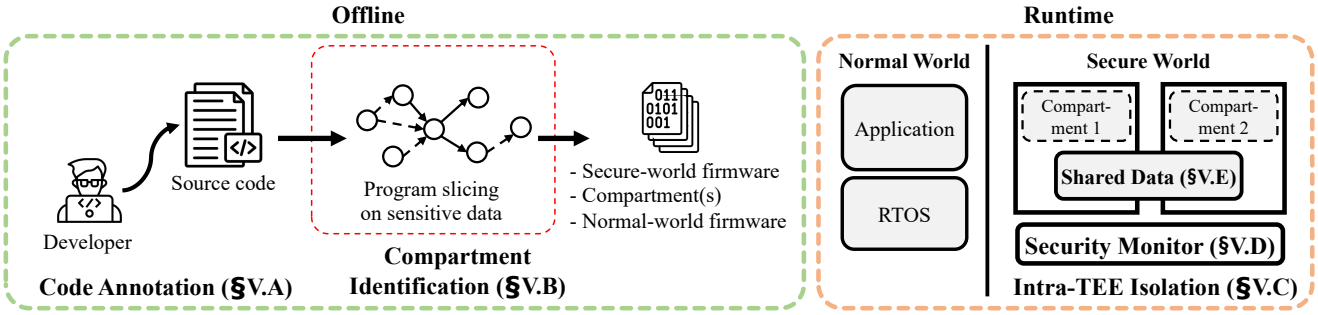


Figure 2: Overview of TZ-DATASHIELD.

Table II: Comparison of different intra-TEE isolation techniques for ARM TrustZone.

Project	Architecture	Enabling Features	Isolation Target
PrOS [15]	ARMv7/8-A	Virtualization	Trusted OS
ReZone [14]	ARMv7/8-A	Hardware-assisted Isolation	Trusted OS
RT-TEE [21]	ARMv8-A/M	Driver Debloating + SFI	Device Drivers
TZ-DATASHIELD	**ARMv8-M**	**SFI + CFI + DFI**	**Sensitive Data Flow**

only. As shown in Table II, solutions like PrOS [15] and ReZone [14] are designed for the ARMv7-A and ARMv8-A architectures, leveraging platform partition controllers (PPCs) such as a secure memory management unit (sMMU) and a resource domain controller (RDC), which are absent in the ARMv8-M architecture used in MCUs. Unlike our approach to isolating the data flows of sensitive data, RT-TEE [21] isolates unverified device drivers within the secure world, similar to component-based compartmentalization [10], [11].

Secure Access to Shared and External Resources. Similar to other compartmentalization techniques, an SDF compartment may have to share data with another compartment to handle sensitive data inevitably. For instance, in PinLock, both `match()` in Compartment 1 and `irq_handler()` in Compartment 2 need to access `key_received` as a shared data (Figure 1(c)). Such shared data can be targeted by an attack that compromises one compartment, say Compartment 2, and manipulates the shared data to affect the execution of the other compartment which relies on the shared data, *e.g.*, corrupting `key_received` to prevent Compartment 1 from unlocking the door even when the correct PIN is entered.

An intra-TEE isolation technique does not prevent such an attack because such shared data is not subject to isolation and should be accessible from the corresponding compartments for legitimate computation. Similarly, peripherals' location cannot be adjacent to the compartments in the address space, such as a peripheral-mapped memory region (*e.g.*, `GPIO` for Compartment 1 and `USART` for Compartment 2) may be maliciously accessed or manipulated by an attacker despite the intra-TEE isolation in place. To mitigate this risk, which prior compartmentalization work [9], [10], [12], [11], [21] does not fully solve, we develop a lightweight mechanism to validate the control and data flow integrity (CFI and DFI) of the compartment execution whenever shared or peripheral data is accessed, preventing advanced attacks from accessing the data, using return-oriented programming (ROP) [22], [23] and data-only attacks [24], [25] for example.

IV. THREAT MODEL

TZ-DATASHIELD aims to protect the confidentiality and integrity of sensitive data in MCU firmware against a strong adversary capable of compromising any software in the normal world, including privileged software such as an interrupt handler in bare-metal firmware or exploiting a control hijacking vulnerability in an RTOS. The adversary's objective is to either steal or tamper with sensitive data via malicious memory or peripheral accesses, targeting full or partial information of the data in the data flow. Even though TZ-DATASHIELD relocates all computations dependent on sensitive data to the secure world, vulnerabilities in these computations can still be exploited due to the lack of internal isolation within the secure world. Such vulnerabilities could serve as a single point of failure, jeopardizing the entire system since ARM TrustZone does not provide an internal isolation mechanism within the TEE, allowing any secure-world program to access any resources in the secure world and normal world. Additionally, the adversary may target shared or peripheral data, which are legitimately

accessible by compartments, to circumvent the compartment isolation mechanism in the secure world.

TZ-DATASHIELD relies on ARM TrustZone and its security features, including secure booting. Additionally, we trust the correctness of TZ-DATASHIELD's software modules and assume they are not vulnerable, including the compiler tool and the security monitor (detailed in §V). The integrity of the security monitor can be verified using the TrustZone secure booting feature. We assume that the firmware source code is accessible. We further assume information flow (CFI and DFI) is an effective mechanism in distinguishing legitimate access from malicious access to shared data (additional discussion in §IX). Our design does not require hardware support for privileged and unprivileged mode separation and memory isolation units (*e.g.*, MPU and PPC) in the secure world.

We do not consider physical attacks, including snooping or manipulating the hardware resources. By relying on the trust model of TrustZone, we do not address the security vulnerabilities inherent to TrustZone hardware, such as side-channel vulnerabilities [26], [27]. Denial-of-service attacks aimed at compromising the availability are out-of-scope.

V. DESIGN

Figure 2 presents an overview of our design. We first describe the offline processes: the firmware code annotation process, where developers annotate the source code to identify sensitive data (§V-A), and the compartment identification process (§V-B), in which our compiler tool statically analyzes the annotated code to find the distinct data flows of the sensitive data. We then explain our runtime mechanisms for intra-TEE isolation (§V-C) and the security monitor to enforce isolation policies within the secure world (§V-D), designed to prevent leakage and unauthorized access between compartments and bypassing the security monitor. Lastly, we detail our strategies used to prevent malicious access to shared data across different compartments and peripheral data (§V-E).

A. Firmware Code Annotation

Manually developing a trusted application to protect data paths is time-consuming and error-prone. This process necessitates that developers have a comprehensive understanding of all data flows across functions and software layers, including third-party libraries. To mitigate this challenge, TZ-DATASHIELD provides a set of easy-to-use C/C++ preprocessors that enable firmware developers to annotate source code efficiently. Using the preprocessors, the developers only need to annotate the data variables or addresses in the source code where the sensitive data are initialized or allocated. TZ-DATASHIELD automatically identifies and isolates the sensitive data flows across all execution paths to and from the annotated variables/addresses, alleviating the need for the developer to annotate every related variable across functions and software layers. Developers can enforce three types of security policies on sensitive data: **(1)** confidentiality protection using `TZDS_*_R`, **(2)** integrity protection using `TZDS_*_W`, and **(3)** confidentiality and integrity protection using `TZDS_*_RW`,

```
/* Global data, confidentiality protection */
const uint8_t key_stored[KEY_SIZE] TZDS_DATA_R = {0x...};
void func() {
  /* Stack data, integrity protection */
  uint8_t buffer[BUFFER_SIZE] TZDS_DATA_W = {0x0};
  ...
}
/* Heap data, confidentiality and integrity protection */
static void *m_head TZDS_HEAP_RW = malloc(...);
/* Peripheral data at [GPIO_BASE,GPIO_BASE+0x1000),
   integrity protection */
#define GPIO_BASE (0x4008C000u)
TZDS_MMIO_W(GPIO_BASE, 0x1000)
GPIO_Type *gpio = (GPIO_Type *) GPIO_BASE;
```

Listing 2: An example code snippet to show code annotation using TZ-DATASHIELD for sensitive data protection.

where `*` can be replaced with `DATA`, `HEAP`, or `MMIO`. TZ-DATASHIELD supports the protection of global and stack data (`DATA`), heap data (`HEAP`), and peripheral data (`MMIO`).

Listing 2 shows how to annotate sensitive data using TZ-DATASHIELD. In this example, the confidentiality of global data in `key_stored` with `TZDS_DATA_R` and the integrity of stack data in `buffer` with `TZDS_DATA_W` will be protected from unrelated code in the normal world and secure world. In addition, the heap data pointed to by `m_head` are annotated with `TZDS_HEAP_RW`, thus confidentiality and integrity are protected. For each pointer variable annotated with a `TZDS_HEAP` preprocessor, TZ-DATASHIELD traces the heap memory allocation and free of the data pointed by the pointer and isolates its data flow from other code. Lastly, the integrity of peripheral registers mapped at `[0x4008C000, 0x4008D000]` will be protected via the `TZDS_MMIO_W(base, size)` annotation. Unlike other preprocessors, the developers need to explicitly specify the base and size of the region to be protected since the memory range may not be statically identifiable in some firmware (*e.g.*, a computed range by arithmetic operators).

B. Compartment Identification

Given the annotated source code of the firmware, TZ-DATASHIELD performs data flow analysis on the LLVM IR code to identify compartments. It first generates a Value Flow Graph (VFG), which statically represents both the control and data dependence of a program. It serves as the input for our compartment identification algorithm, the pseudo-code of which is presented in Algorithm 1. The output of the algorithm is compartment identification information which is essential for the compiler to generate images for the compartments, secure-world firmware, and normal-world firmware.

Sensitive Data Flow. For each sensitive data requiring *confidentiality protection* (*i.e.*, annotated using a `TZDS_*_R` or `TZDS_*_RW`), TZ-DATASHIELD conducts *forward slicing* (line 10) based on data flow analysis [28] to identify instructions that directly reads the data and data objects that are potentially influenced by the data (line 4). In addition, it conducts a context-sensitive point-to analysis [28] to find instructions that indirectly read the data through aliased pointers. Similarly, for each sensitive data requiring *integrity protection* (*i.e.*, annotated using a `TZDS_*_W` or `TZDS_*_RW` preprocessor),

Algorithm 1: Sensitive Data Flow Compartment Identification.

Input : Value flow graph (VFG) - VFG.
Input : A set of sensitive data to protect - S_n.
Output: A set of SDF compartments – C_n.

```
1  C_n ← ∅
2  V_D_n ← ∅
3  foreach V ∈ VFG do
4      if V.address ∈ S_n then
5          VD_n.add(V)
6      end
7  end
8  foreach V ∈ VD_n do
9      if TZDS_R ∈ V.mark then
10         SF_n.add(getForwardSlices(V))
11     end
12     if TZDS_W ∈ V.mark then
13         SB_n.add(getBackwardSlices(V))
14     end
15 end
16 foreach slice ∈ SF_n ∪ SB_n do
17     Fn_slice ← group nodes in slice by function
18     foreach Fn ∈ Fn_slice do
19         Fn.sensitive_data.add(slice.sensitive_data)
20     end
21 end
22 foreach S ∈ S_n do
23     C ← ∅
24     foreach Fn ∈ all functions do
25         if Fn.sensitive_data.size() = 1 and
             S ∈ Fn.sensitive_data then
26             C.add(Fn)
27         end
28     end
29     C_n.add(C)
30 end
31 foreach Fn ∈ all functions do
32     if Fn.sensitive_data.size() > 1 then
33         C_n.add(C)
34     end
35 end
36 return C_n
```

the compiler tool conducts *backward slicing* (line 13) to find instructions and data objects that can potentially influence the sensitive data, either directly or indirectly.

As a result of this analysis, each SDF compartment is assigned a minimal set of code, data, and peripheral-mapped memory regions which encompasses a distinct data flow of the sensitive data. More specifically, an SDF compartment only contains the result of forward slicing on one sensitive data as the source for confidentiality protection, or the result of backward slicing on one sensitive data as the sink for integrity protection. If both the confidentiality and integrity need protection (annotated using a `TZDS_*_RW` preprocessor), TZ-DATASHIELD performs a union of the forward and backward slicing results to generate a compartment for reading and writing the sensitive data. Overall, this compartment granularity based on sensitive data flow ensures that only minimal computing resources will be assigned to each compartment, reducing both the attack surface and the frequency of the compartment switching compared to existing compartment units, as experimentally demonstrated in §VII.

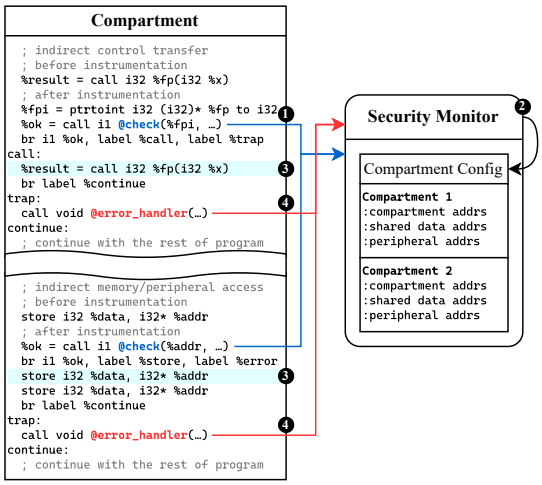
Shared Code and Data. Since different compartments may need to execute common code (*e.g.*, a library or RTOS function) and communicate through shared data (*e.g.*, global variables), TZ-DATASHIELD compares the data dependency of every compartment and identifies the shared code and data across them. Each of these shared code and data objects is then relocated into a separate ELF section, and these compartments are instrumented to access them via the compartment isolation boundary securely. At runtime, our intra-TEE isolation mechanism ensures that the shared code is isolated using SFI, while access to the shared data and peripheral data is additionally validated using CFI and DFI enforcement (§V-E).

Using the result of the above compartment identification steps, TZ-DATASHIELD compiles the IR code and links the binary objects to generate the images of the SDF compartments, secure-world firmware, and normal-world firmware. During this process, it creates three distinct sections for each SDF compartment: code section, private data section, and metadata section to contain the information of the shared code, shared data, and peripheral data. These compartment sections are added to the normal-world firmware image, so that they are kept outside the secure world while not being executed, minimizing the *temporal attack surface*. The secure-world firmware includes the security monitor that contains auxiliary routines to securely load the compartments from the normal world to the secure world, and execute them. The normal-world firmware includes the rest of the original firmware that does not need access to the sensitive data.

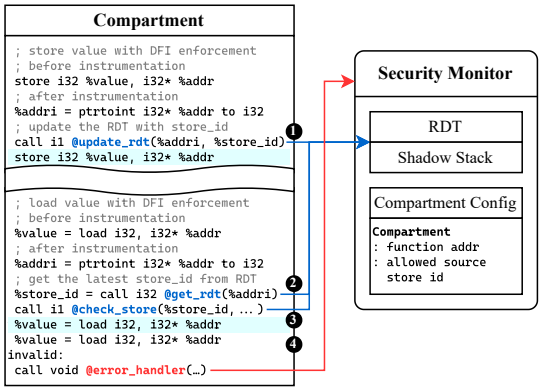
C. Intra-TEE Compartment Isolation

We leverage software fault isolation (SFI) for intra-TEE isolation, to avoid known hardware limitations of memory isolation in MCUs (refer to §III). It consists of a compile-time instrumentation module and runtime checks performed by the security monitor. For each compartment, the compiler generates three distinct sections: code, private data, and metadata sections. The code section is loaded into read-only memory (*e.g.* flash memory), while the private data section is loaded into read-write memory (*e.g.* SRAM). Besides, the metadata section specifies a list of the shared code, shared data, and peripheral data for the compartment. Given LLVM IR instructions, the instrumentation ensures that all instructions for data access target the appropriate address ranges of the private data section, shared, or peripheral data referring to the metadata, and all control-flow transfer instructions target a list of function entry addresses in the code section.

Since our trusted compiler ensures that all *direct* data accesses and control transfers are correct, only *indirect* data accesses and control transfers need to be instrumented. This guarantees that each compartment operates within its designated areas, preventing unauthorized instructions and maintaining isolation within the secure world. Figure 3(a) illustrates how TZ-DATASHIELD instruments indirect control transfers (in Compartment 1) and data accesses (in Compartment 2). TZ-DATASHIELD identifies all instructions for indirect control transfers (*e.g.* `call` and `invoke`) and inserts a checking routine before them. During runtime, this routine verifies whether the transfer target address is within the compartment's address



(a) SFI enforcement.



(b) CFI and DFI enforcement.

Figure 3: LLVM IR instrumentation for (a) SFI enforcement on an indirect control transfer and memory access, and (b) CFI and DFI enforcement on shared data accesses.

range. Similarly, TZ-DATASHIELD inserts a checking routine of the security monitor for all instructions involving indirect data accesses (*e.g.*, `load` and `store`). The security monitor ensures that the access target is either within the private data section or the allowed data list.

In TZ-DATASHIELD, every compartment and security monitor operates in the privileged mode within the secure world. This arrangement prevents frequent context switches between the security monitor and the compartments via SVC, thereby improving performance. Note that SFI also shields the security monitor from unauthorized access from compartments.

D. Security Monitor

The security monitor enforces the correctness of secure booting, SAU configuration, and compartment execution.

Secure Boot. The secure reset interrupt handler is the entry point of the secure world firmware. When an MCU is powered on, the secure bootloader of TrustZone jumps to the secure reset interrupt handler, which then verifies the integrity of the security monitor. This prevents any strong attacker in the normal world from tampering with the security monitor.

Security Attribution Unit (SAU). The SAU is configured to allocate different memory regions. Each SAU region can be configured as Secure (`S`), Non-Secure (`NS`), or Non-Secure Callable (`NSC`). The `S` and `NS` regions are designated for the secure and normal worlds respectively, while NSC regions facilitate cross-world invocations with a special instruction called Secure Gateway (`SG`). TZ-DATASHIELD uses three S regions for its secure-world firmware: the security monitor code, the running compartment, and the shared data and the stack/heap memory. Within a compartment, its peripheral region in the secure world is also configured as an S region. The secure world firmware also assigns one NSC region containing routines to load and execute compartments. Once the initialization is completed, the control is handed over from the secure world to the normal world.

Compartment Execution. The security monitor mediates cross-world invocations. When a compartment is invoked, it initializes the required RAM regions and then loads the corresponding compartment. Although TZ-DATASHIELD loads compartments on-demand by default, developers have the option to load all the compartments altogether at boot time for efficiency. For security, TZ-DATASHIELD is integrated with heap management in the security monitor to handle heap memory usage of itself and compartments. For SFI, CFI, and DFI, the security monitor has to maintain an up-to-date list of dynamic memory allocations allowed for each compartment; further details on CFI and DFI are provided in §V-E. Specifically, the security monitor tracks all heap allocations and free at runtime, associating each with its owner compartment. Likewise, it updates the address range of stack frames upon function calls and returns.

Additionally, compartment execution must be able to respond to interrupts. Since interrupts are triggered by hardware, the corresponding interrupt handler may not reside within the currently active compartment. In such cases, the security monitor should update the context of the newly active compartment and mediate the compartment switch. To enforce comprehensive security checks, TZ-DATASHIELD implements a secure interrupt dispatcher for each interrupt handler in the secure world, registering this dispatcher in the interrupt vector table (IVT). When an interrupt occurs, the hardware triggers the dispatcher, which then redirects control to the original handler. The secure world IVT is immutable and stored in read-only flash memory. All addresses within the IVT point to a common dispatcher. This dispatcher forwards requests to the security monitor, which in turn invokes the actual handler within the isolated compartment.

E. Shared and Peripheral Data Protection

Although an individual SDF compartment may have its private memory/peripheral data, multiple compartments may need to share data. As discussed in §III, such shared and peripheral data are subject to potential attacks that try to access the data for stealing and manipulation since access to these data should be allowed for the compartments to do their legitimate job, thereby cannot be blocked by compartment

isolation. We address this issue by validating the context in which the shared and peripheral data is accessed. Specifically, we enforce lightweight CFI and DFI on the execution path that accesses the shared resource. In contrast, access to shared code does not need additional validation since exploiting it does not enable an attacker to compromise the confidentiality or integrity of data in other compartments.

Lightweight CFI and DFI Enforcement. We develop a lightweight mechanism to validate the CFI and DFI of the compartment execution that attempts to access shared or peripheral data. For any indirect data access or control transfer instruction, our intra-TEE isolation mechanism first performs an SFI check to confine the target within the memory ranges of the compartment. TZ-DATASHIELD additionally performs CFI and DFI checks if and only if the target is one of those shared or peripheral data assigned to the compartment. Figure 3(b) illustrates the CFI and DFI enforcement. Specifically, the CFI enforcement ensures that prior indirect control transfer targets that have been recorded are valid while the DFI enforcement checks if prior indirect memory accesses were performed by valid `store` and `load` instructions. A CFI check compares each control transfer target with the valid targets statically identified during the compartment identification including `call` and `ret` instructions. To verify the target of a return instruction, our security monitor maintains a shadow stack to keep track of the return address at runtime (`ret`). For DFI enforcement, each `store` instruction that writes to the shared or peripheral data is assigned with a unique `store_id`, and each `load` instruction is assigned with a list of allowed `store_id`s during the static analysis. At runtime, the security monitor maintains a *runtime definitions table* (RDT) that records the `store_id` of the last `store` instruction that accesses shared or peripheral data, and before each `load`, the security monitor checks if the `store_id` is in the allowed `store_id` list.

The sizes of the shadow stack and RDT maintained by the security monitor are configurable, allowing them to store the last N targets of control transfers or data accesses for each SDF compartment. When compartment access to shared or peripheral data is detected, these N targets are validated using the inter-procedural control graph (ICFG) and value flow graph (VFG) constructed during the compartment identification, by matching them with the valid targets found statically. Since our CFI/DFI mechanism is only activated when there is shared or peripheral data access, which is infrequent, only a minimal overhead is incurred (see §VII-C1).

VI. IMPLEMENTATION

TZ-DATASHIELD is prototyped in approximately 1.2K lines of C (security monitor), 1.7K lines of C++, and 2.8K lines of Python code (static analysis). We developed TZ-DATASHIELD using Clang/LLVM-14 and the SVF library [28] for the inter-procedural value flow and points-to analyses. The C/C++ preprocessor for code annotation is implemented using macros of `__attribute__` provided in a header file. We implemented LLVM passes that identify sensitive data flows, conduct program slicing, and instrument LLVM IR code for SFI, CFI, and DFI. Since the default linker in the LLVM toolchain does not support ARM TrustZone, we use a GNU BFD linker to generate the firmware images. The compartments are generated as position-independent code (PIC), allowing them to be loaded at any memory location. To dynamically generate a linker script for each compartment, we use a Python script to parse the compartmentalization results and specify their memory layout.

VII. EVALUATION

In this section, we evaluate the security and performance of TZ-DATASHIELD prototype using a set of real-world applications. First, we measured the security impacts of TZ-DATASHIELD by quantifying the number of secure-world address space and ROP gadgets reduction in the firmware. Then, we analyzed how TZ-DATASHIELD can prevent attackers from exploiting the vulnerabilities in a compartment to steal or tamper with sensitive data. We finally measured the performance overhead by comparing the runtime and memory footprint of the applications with and without TZ-DATASHIELD. Additionally, we conduct formal verification of the TZ-DATASHIELD security monitor and evaluate the developer effort necessary for integrating TZ-DATASHIELD into applications. Furthermore, we measure other performance-related metrics, including the number of compartment switches and SFI/CFI/DFI checks, as detailed in Appendix §A.

Experimental Setup. We evaluate security impacts and performance overhead imposed on twelve real-world MCU applications. The experiments were conducted on the LPCXpresso55S69 [17] development board, featuring an LPC55S69 MCU with an ARM Cortex-M33 processor based on the ARMv8-M architecture. This MCU operates at 150 MHz and is equipped with 640 KB flash and 320 KB RAM.

Twelve applications are used in our evaluation – six applications with RTOS and six bare-metal ones without RTOS. The applications running on RTOS exhibit more complex data flows, derived from OS functions such as semaphores used to handle interrupt signals. Additionally, RTOS supports threading, which facilitates the demonstration of thread-granularity compartmentalization. The applications are as follows:

(1) PinLock and **PinLock-FreeRTOS** simulate a smart lock by controlling a motor according to the PIN entered via USART; **(2) Temp** and **Temp-FreeRTOS** read a temperature sensor connected to an ADC; **(3) Accel** and **Accel-FreeRTOS** receive accelerometer data via I2C bus and transmits the results via USART; **(4) Gyro** and **Gyro-FreeRTOS** receive gyroscope data through an SPI bus and transmit it via USART; **(5) SD-FatFS** and **SD-FatFS-FreeRTOS** mounts a file system on SD card insertion, and intensively test file operations in a loop; **(6) USBVCom** and **USBVCom-FreeRTOS** emulates a USB to a serial port device which facilitates the connection of a serial port to a desktop computer.

We utilize PinLock [12], a system typically found in IoT. All other applications are selected from the NXP MCUXpresso SDK [29] to cover a wide range of functionalities and complexity levels, including peripherals, communication protocols,

Table III: Bare-metal and RTOS applications with various sensitive data protected using different compartmentalization techniques in comparison. The table shows the sensitive data objects annotated in the firmware code (**Sensitive Data**), number of source lines of firmware code (**#LoC**), number of identified compartments (**#C**), average number of functions per compartment (**#Fn**), and average size of the compartment (**Size**) under each compartmentalization technique. The results for the bare-metal applications with thread-based compartments are unavailable due to the absence of an OS in the firmware.

	Application	Sensitive Data	#LoC	Sensitive Data Flow			Function		Component			Thread		
				#C	#Fn	Size (KB)	#Fn	Size (KB)	#C	#Fn	Size (KB)	#C	#Fn	Size (KB)
Bare-metal	PinLock	key, buf, USART, GPIO, HASHCRYPT	43.5K	**15**	**2.3**	**0.55**	34	0.35	4	8.5	1.57	-	-	-
	Temp	temp, completed_flag, USART, ADC0	53.2K	**14**	**4.8**	**1.45**	67	0.45	2	33.5	9.05	-	-	-
	Accel	r_buffer, x, y, z, USART, I2C	53.7K	**6**	**11.8**	**3.28**	71	0.45	2	35.5	9.48	-	-	-
	Gyro	x, y, z, USART, SPI, PIN_INT	54.9K	**14**	**5.2**	**1.63**	73	0.46	3	24.3	6.93	-	-	-
	SD-FatFS	w_buffer, r_buffer, USART, SDIO	84.6K	**6**	**34.5**	**8.39**	207	0.43	5	41.4	10.03	-	-	-
	USBVCom	cdc_vcom_handler, USART, USB	65.4K	**7**	**18.4**	**5.57**	129	0.48	2	64.5	19.03	-	-	-
FreeRTOS	PinLock	key, buf, USART, GPIO, HASHCRYPT, usart_sem, gpio_sem	68.2K	**17**	**4.5**	**0.62**	77	0.28	4	19.3	2.02	10	7.7	0.92
	Temp	temp, completed_flag, USART, ADC0, adc_sem	78.1K	**15**	**7.3**	**1.51**	110	0.37	3	36.7	6.77	11	10.0	1.98
	Accel	r_buffer, x, y, z, USART, I2C, i2c_sem	78.7K	**7**	**16.3**	**3.14**	114	0.37	3	38.0	7.07	5	22.8	4.32
	Gyro	x, y, z, USART, SPI, PIN_INT, spi_sem	80.0K	**15**	**7.7**	**1.67**	116	0.38	4	29.0	5.76	12	9.7	2.05
	SD-FatFS	w_buffer, r_buffer, USART, SDIO, sem	109.4K	**7**	**35.7**	**7.52**	250	0.40	4	62.5	13.02	12	20.8	4.47
	USBVCom	cdc_vcom_handler, USART, USB, usart_sem, usb_sem	90.2K	**8**	**21.5**	**5.16**	172	0.42	4	43.0	10.13	11	15.6	3.80

Table IV: Number of shared data objects identified by different compartmentalization techniques and their combined sizes.

	Application	Sensitive Data Flow		Function		Component		Thread	
		#	Size (B)	#	Size (B)	#	Size (B)	#	Size (B)
Bare-metal	PinLock	**4**	**119**	16	343	6	155	-	-
	Temp	**4**	**25**	29	425	3	41	-	-
	Accel	**5**	**29**	31	429	5	45	-	-
	Gyro	**4**	**26**	33	486	6	42	-	-
	SD-FatFS	**5**	**1830**	86	12006	4	1954	-	-
	USBVCom	**7**	**1372**	59	16572	5	1664	-	-
FreeRTOS	PinLock	**6**	**123**	59	767	7	151	6	151
	Temp	**5**	**29**	29	845	4	41	7	57
	Accel	**6**	**33**	74	841	6	45	8	45
	Gyro	**5**	**26**	76	886	7	38	7	54
	SD-FatFS	**6**	**1834**	129	12490	5	1958	5	2014
	USBVCom	**8**	**1376**	102	17056	6	1668	5	1786

and file system operations. We add FreeRTOS integration for some of them without changing the semantics.

We conduct experiments to compare our compartmentalization approach with existing ones, which compartmentalize the firmware at function-, component-, or thread-granularity. We implement these compartmentalization policies based on the conceptual design of previous works [12], [9], [10], [11]. To this end, we implemented four compartmentalization policies: SDF (our design), thread, function, and component. We have annotated sensitive data to be protected using the provided pragma directives. The results are summarized in Table III.

A. Quantitative Security Analysis

We evaluate TZ-DATASHIELD using the following quantitative metrics:

- **Secure-World Address Space:** Isolating a compartment restricts an adversary's capability to conduct attacks that access resources (*i.e.*, code, private/shared data, and peripheral data) outside the compartment. This metric measures the size of the address space a compartment can access.
- **Number of Return-Oriented Programming (ROP) Gadgets:** This metric quantifies the code fragments ending with "return", namely ROP gadget. More ROP gadgets mean the firmware is more susceptible to ROP attacks. We used Ropper [30] to count the number of ROP gadgets.
- **Accuracy of Static Analysis:** This metric measures over-approximation and under-approximation in terms of data flow (see Appendix §A-C).

1) Secure-World Address Space Reduction: Figure 4 shows the average size of secure-world address space that a compartment can access for different compartmentalization units. As summarized in Table V, the SDF compartmentalization reduces the secure-world address space accessible by a compartment by 85.6% for code, 96.4% for shared data, 98.2% for peripheral data, and 80.8% in total. Compared to other compartmentalization units, SDF compartmentalization achieves the highest reduction in shared data due to fine-grained data flow tracking. Although the function-based compartmentalization can further reduce the code and peripheral exposure by 98.4% and 99.7%, it exhibits security risks due to an increase in the shared data exposure.

2) ROP Gadget Reduction: Figure 5 shows the numbers of ROP gadgets found in baseline implementation (*i.e.*, without intra-TEE isolation), and solutions with compartmentalization enforced at SDF, function, component, and thread granularity. The last column of Table V shows that the sensitive data flow units can reduce the number of ROP gadgets by 88.6% on average. Compartmentalization in function granularity can further reduce the number of ROP gadgets by 98.6% on average; however, as aforementioned, it incurs more security risks and performance overheads. Note that an attacker may exploit available ROP/JOP gadgets within a compromised compartment. However, it is still isolated within the compartment due to CFI/DFI enforcement.

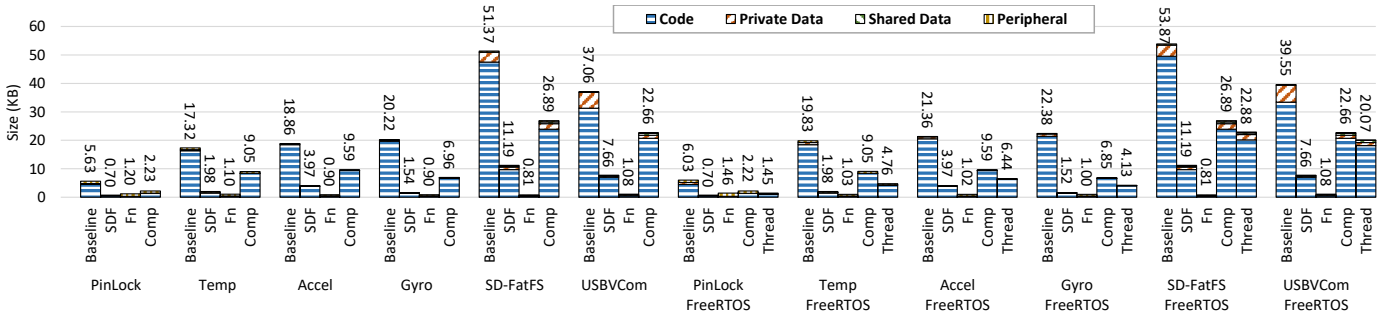


Figure 4: Average size of secure-world address space accessible by no-intra-TEE isolation firmware and compartments based on different compartment units: sensitive data flow (**SDF**), function (**Fn**), component (**Comp**), and thread (**Thread**).

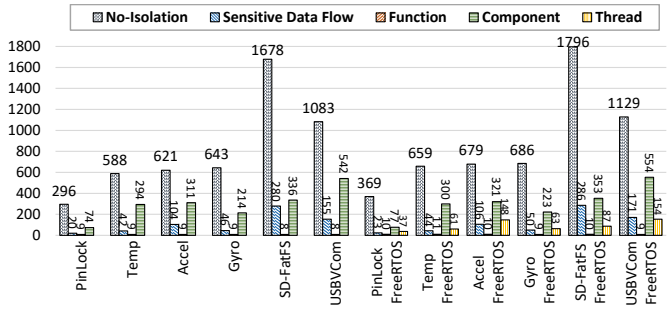


Figure 5: Average number of ROP gadgets per compartment.

Table V: Average reduction rates of secure-world address space and ROP gadgets per compartment achieved by different compartmentalization techniques.

Unit	Address Space Reduction				ROP Gadgets
	Code	Shared Data	Peripheral Data	Total	
SDF	**85.6%**	**96.4%**	**98.2%**	**80.8%**	**80.8%**
Fn	98.4%	79.2%	99.7%	96.2%	98.6%
Comp	61.1%	94.6%	94.0%	38.4%	63.1%
Thread	77.8%	94.8%	96.3%	62.7%	88.5%

B. Qualitative Security Analysis

Assuming that attackers can exploit the vulnerabilities in a compartment, this section discusses what attackers can achieve. For example, CVE-2018-16528, CVE-2018-16525, and CVE-2018-16526 are buffer overflow vulnerabilities in the FreeRTOS and CVE-2021-42553 are in the USB library for MCU. CVE-2019-7711, CVE-2019-7712, and CVE-2019-7715 are format string vulnerabilities found in MCU software. In particular, we are interested in whether the compromised compartment can be utilized to **(1)** steal or modify private data of itself and other compartments, **(2)** steal or modify the content of shared data, **(3)** control or steal data from peripherals, **(4)** break the security guarantees of TZ-DATASHIELD's security monitor. We experimentally show that these CVEs can be confined within their affected compartments, instead of being exploited to reveal to tamper with sensitive data.

1) Shellcode Injection Attack: Shellcode injection attacks can be launched by exploiting software vulnerabilities, such as buffer overflow and format string vulnerabilities to execute arbitrary code. The attacker can steal or modify the private data of the compromised compartment or other compartments by injecting store/load instructions to access these data or necessary code to utilize existing instructions inside the compromised compartments. Even worse, the attacker can modify the data of the security monitor to compromise the whole system utilizing the injected code. However, TZ-DATASHIELD ensures that the code area is not writable and the data area is not executable. Also, our SFI mechanism prevents the attacker from accessing the data of the security monitor.

2) Code-Reuse Attack: The code-reuse attack is a type of exploitation where an attacker leverages existing code within the firmware to perform malicious actions, circumventing traditional security measures like data execution prevention. An attacker can hijack the control flow by overwriting the return address on the stack or tampering with function pointers. The attacker can also reuse the code of other compartments that contain shared data or peripheral accessing instructions to steal or modify data. However, the SFI mechanism can block the compromised compartment from accessing the code of other compartments. As described in §V-C, TZ-DATASHIELD has a shadow stack to ensure the integrity of return addresses and the modification of the function pointers can be detected before the call due to CFI violation.

3) Data-Only Attack: Data-only attacks [24], [25] focus on manipulating the sensitive data within an application to affect its behavior without hijacking the control flow. The attacker may utilize a compromised compartment to manipulate the value of shared data used in conditional checks (*e.g.* if-else statements), thereby altering the system's behavior to bypass security measures or initiate unauthorized actions. Additionally, the attacker may directly manipulate the output of a peripheral to execute malicious operations. The attacker may also directly manipulate the output of a peripheral to perform malicious operations. Unlike attacks that hijack the control flow, data-only attacks, which do not alter the control flow but disrupt legitimate data flow, are difficult to detect with CFI or SFI. However, the manipulation of the shared or peripheral data can be defeated by our DFI mechanism.

4) IRQ Handler Attack: The IRQ handler is a special type of function that is called when an interrupt is triggered by

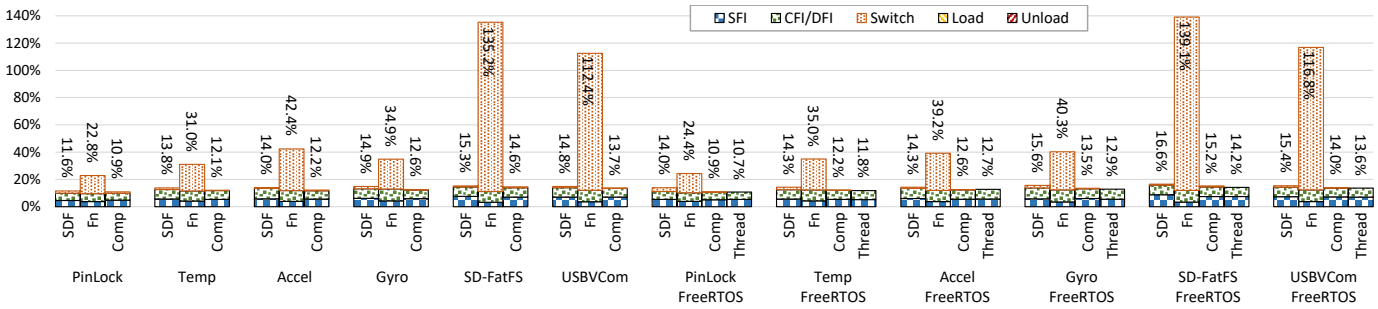


Figure 6: The runtime overhead of bare-metal and RTOS applications with different compartment units. CFI/DFI enforcement is enabled in all compartmentalization techniques with one target validated at every shared and peripheral data access (N=1).

Table VI: Micro-benchmark results of TZ-DATASHIELD. SFI and CFI/DFI latency are for one target address to validate.

Component	Min. (μs)	Max. (μs)	Avg. (μs)
SFI	0.21	2.20	0.72
CFI/DFI (N=1)	0.17	0.31	0.19
Compartment Switch	2.13	2.66	2.28
Compartment Load	0.39	0.58	0.45
Compartment Unload	0.26	0.47	0.33

Table VII: Average memory footprint (KB) of sensitive data protection with different compartmentalization techniques.

		SDF	Function	Component	Thread
Flash Memory (Code)	Security Monitor	13.98	23.60	12.99	13.21
	Compartments	21.66	35.23	20.25	14.96
	Normal World	10.98	10.97	10.83	15.12
	Total	**46.62**	**69.81**	**44.07**	**43.29**
SRAM (Data)	Security Monitor	4.82	9.35	4.35	4.46
	Compartments	0.85	0.85	0.85	0.49
	Normal World	1.32	1.32	1.32	1.56
	Total	**6.99**	**11.51**	**6.52**	**6.51**

hardware. The attacker can exploit this to access other compartment resources from the IRQ handler of a compromised compartment because the security monitor does not have a chance to securely handle the compartment switching when a hardware IRQ is invoked. However, we address this issue by developing a secure interrupt dispatcher, as described in §V-D.

C. Runtime Overhead

1) Application Benchmarks: We measured the runtime overhead of Twelve applications. The results (Figure 6) detail the overhead contributions from SFI, CFI/DFI, compartment switch, compartment load, and compartment unload. Overall, the average overhead over all applications is imposed by load/unload/switch (1.4%), SFI (6.3%), and CFI/DFI (7.0%), total 14.7%. The results indicate that the overhead is significantly influenced by the compartment unit, regardless of the applications. In most cases, the function unit incurs the highest overheads due to frequent compartment switches. As compartment granularity increases, the overheads for compartment switch, load, and unload increase, and the overheads for SFI and CFI/DFI decrease. Similarly, the component and thread units show low overhead in the compartment but higher security enforcement overhead as these compartment units have a larger memory footprint. For instance, the *Accel* application experiences 5.6% and 6.1% overhead for SFI and CFI/DFI, and 0.5% overhead for the compartment switch when using the component compartment. With function compartment units, however, the overheads are 3.9%, 7.9%, and 30.6%, respectively. Compared to component and function compartments, SDF compartments achieve 5.8%, 7.6% overhead for SFI and CFI/DFI, and 0.7% overhead for the compartment switch. The SDF compartments provide better security with comparatively moderate overheads.

2) Micro-benchmarks: To further understand the runtime overhead incurred by the CFI/DFI enforcement, we measured the overhead associated with different values of N. N is a configurable number that controls the number of control transfer and data access targets to be validated by the security monitor. Figure 8 demonstrates that the overheads increase almost linearly with N. For instance, with our compartmentalization based on sensitive data flow, the overhead is 6.90% when N=1 and it increases roughly 1.4% whenever N increments. The difference between the overheads of different compartment units is due to the compartment switch – the checking mechanism needs to take more steps to ensure the control flow in the compartment level, making sure it is a legal compartment invocation, and this checking is mandatory and not controlled by N. We also measured the time to take one operation for the SFI check, the CFI/DFI check, the compartment switch, the compartment load, and the compartment unload as shown in Table VI. The results show that the time consumption of SFI checks varies due to the difference in compartment memory footprint, while the compartment switch is relatively stable. Note that in our application benchmarks, the compartment load/unload overhead is zero as the compartments are loaded during boot time and the RAM space in the secure world is large enough to accommodate all compartments.

D. Memory Overhead

We evaluate the memory overhead of TZ-DATASHIELD by computing the size of different sections of compartments, the

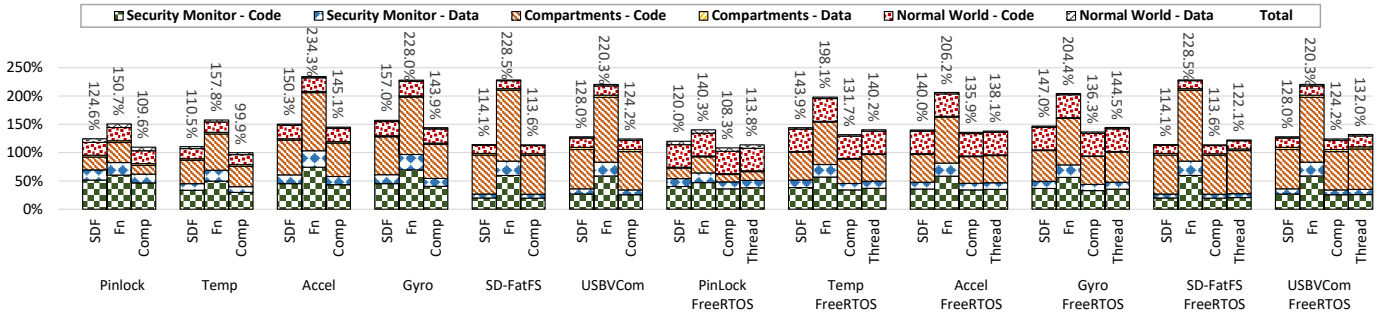


Figure 7: Memory footprint of sensitive data protection using different compartmentalization techniques (size is normalized to the unprotected firmware).

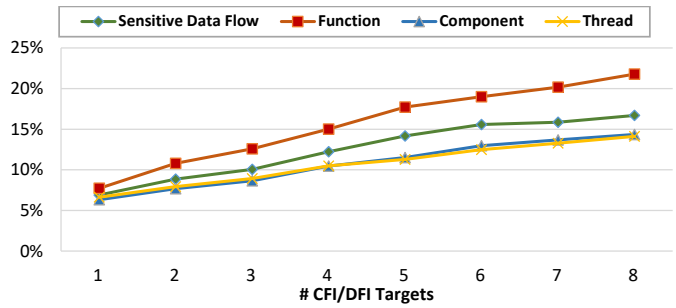


Figure 8: Runtime overhead incurred by the CFI/DFI enforcement with an increasing number of control transfer and data access targets to validate (N=[1..8]).

security monitor, and normal-world firmware. Table VII shows the memory overhead of TZ-DATASHIELD in comparison with other compartmentalization techniques. We observed that the memory overhead depends on both the application and the selected compartmentalization units. TZ-DATASHIELD increases the memory usage by 16.7 KB for the security monitor and by 136 bytes per compartment to store the metadata of compartments, including the initial allowed memory regions and the position of the compartments. The security monitor also has a 4-KB memory pool used for the dynamic memory allocation of compartments, which is configurable.

For compartments, the memory overhead mainly comes from the code instrumentation to implement SFI, CFI, and DFI. For each indirect memory access or branch instruction, the instrumentation adds a few instructions to check the memory access or branch target. If an application has more indirect memory accesses or branches, the memory overhead will be higher. For example, the *Accel* and *Gyro* applications have more indirect memory accesses due to frequent device communication, leading to higher memory overhead. Another source of memory overhead incurred by compartments is the code to copy data from read-only memory to the compartment's SRAM region. This overhead is proportional to the number of compartments (192 bytes). As a result, the function compartmentalization has a higher memory overhead than other units, as it creates more compartments.

VIII. RELATED WORK

Compartmentalization for MCUs. Compartmentalization techniques [9], [12], [11], [10] are crucial for enhancing the security of MCUs by isolating different compartments. As described in §III, these techniques vary in compartment unit and enforcement mechanisms to meet different requirements. From the perspective of shared data protection, ACES [12] proposes an emulator for stack protection, while EC [10] and CRT-C [11] introduce type extensions in their runtime services for access control over typed (annotated) shared data. Unlike TZ-DATASHIELD, these techniques do not ensure the control and data integrity of the shared data. Similarly, M2Mon [19] and EPOXY [20] leverage static analysis to identify code to be protected and separate the code from others. M2Mon [19] incorporates a reference monitor that intercepts and controls MMIO operations based on registered rules, detecting these operations through static analysis. EPOXY [20] identifies firmware segments requiring privilege mode. It implements a privilege overlay, granting privileges only for the execution of these segments, rather than the entire firmware. While M2Mon and EPOXY separate the target operations from others, TZ-DATASHIELD isolates each compartment from other compartments for sensitive data protection.

Intra-TEE Isolation for ARM TrustZone. Since ARM TrustZone has been widely supported in the ARMv7-A and ARMv8-A architectures, existing works [31], [15], [14] have proposed compartment isolation techniques to enhance the security of software running in the secure world using TrustZone. Specifically, Rubinov *et al*. [31] automatically partitions an Android application into the secure world and the normal world to protect sensitive programs handling confidential data in the secure world as Trust Applications (TAs). Solutions like PrOS [15] and ReZone [14] have been developed to protect TAs by enforcing the isolation of the trusted OS in the secure world. Since there is no isolation mechanism within the secure world of ARM TrustZone, a compromised trusted OS can subvert the entire system. PrOS [15] proposes a software-based virtualization technique for each trusted OS to have its own virtualized TrustZone resources. Similarly, ReZone [14] relies on hardware-based memory isolation such as sMMU.

For MCUs supporting ARM TrustZone, several fault isola-

tion techniques [32], [21], [33], [34] tailored to this environment have been proposed. Pinto *et al*. [32] leverages TrustZone to virtualize multiple cores for Cortex-M devices. RT-TEE [21] is a real-time trusted execution environment to resolve real-time constraints. Specifically, it isolates device drivers and the reference monitor protects the real-time system from compromised drivers within the secure world. Unlike these works, TZ-DATASHIELD focuses on isolating the data flows of sensitive data within the secure world for data confidentiality and integrity.

IX. Discussion and Limitation

Attacks through NSC. A sophisticated adversary with full knowledge of the target firmware may send malicious input from the normal world to an SDF compartment in the secure world through the NSC region. Although TZ-DATASHIELD ensures that such input will not have any data dependency on the sensitive data, the adversary may hijack the execution of the compartment by exploiting a vulnerability in the compartment through the input. In addition, the ARMv8-M architecture suffers from return-to-non-secure vulnerabilities [35], which redirect a compromised control transition instruction in the secure world to malicious code residing in the normal world when exploited. Although preventing such attempts through the NSC region completely is out of our scope, TZ-DATASHIELD confines the impact of such an attack within the compartment, significantly limiting the chance of compromising the sensitive data.

Direct Memory Access (DMA) Attacks. DMA enables direct data transfers from/to peripheral-mapped regions without the intervention of the CPU. This process consists of three steps: **(1)** the processor writes the DMA configuration to the DMA controller, including the memory address to use; **(2)** upon triggering a DMA operation, the DMA controller takes control of the bus to transfer the data; **(3)** once the transfer completes, the DMA controller issues an interrupt to notify the processor. The integrity of DMA operations hinges entirely on the configuration settings. Without security measures for the DMA configuration, attackers could manipulate DMA transfers to access unauthorized memory areas. Although we do not consider these attacks, orthogonal solutions like D-Box [36], which secures the areas storing DMA configurations, can be adopted.

Mimicry/Confused Deputy Attacks. An attacker may exploit a compromised compartment to perform a mimicry attack (*e.g*., control-flow bending) or confused deputy attack by invoking another compartment within the valid control/data flows where the attacker tries to disguise malicious actions as legitimate ones to avoid detection by CFI and DFI checks. However, we note that attackers are limited to performing attacks that do not violate both our CFI and DFI checks. Since attacks that keep both control and data flows intact are challenging, this essentially limits the impact of the attacks to manipulating compartment loading times and compromising availability, which is outside the scope of this work.

Type Confusion/VPTR Overwrite Attacks. For firmware written in C++, attackers may exploit type confusion or virtual table pointer (VPTR) overwrite vulnerabilities to manipulate the control flow of a compartment, bypassing TZ-DATASHIELD's protection. We plan to employ existing solutions to mitigate these issues, such as HexType [37], which are orthogonal to our work.

Over-approximated Compartments. Static program analysis is known to suffer from over-approximation due to the challenges of accurate points-to analysis. Consequently, similar to other compartmentalization techniques based on static analysis [9], [10], [12], [11], TZ-DATASHIELD may inadvertently include extraneous firmware code and data that is unrelated to the sensitive data it aims to protect into a compartment. This could lead to the inclusion of unnecessary variables in the sensitive data flow. Note that this may affect runtime performance, but never poses additional security risks.

X. Conclusion

TZ-DATASHIELD proposes a compiler tool to protect the confidentiality and integrity of sensitive data using ARM TrustZone for MCU-based systems. After developers annotate data to be protected, it automatically identifies the sensitive data flow and generates SDF compartments. By leveraging TrustZone primitives, TZ-DATASHIELD thwarts strong adversaries that exploit vulnerabilities in privileged software. We introduce an intra-TEE isolation mechanism using SFI to provide an isolated execution environment for each compartment in the secure world. Additionally, we propose a lightweight mechanism to validate the control and data flow integrity of compartments that access shared and peripheral data. We implemented a prototype of TZ-DATASHIELD and analyzed its security and performance. Our evaluation shows that TZ-DATASHIELD significantly reduces the attack surface only incurring a minimal performance and memory overhead.

Acknowledgment

We thank the anonymous reviewers for their insightful feedback. This work was supported in part by the Texas A&M Engineering Experiment Station on behalf of its SecureAmerica Institute, Agency for Defense Development (ADD) grant funded by the Korean government (U23056TF), and U.S. National Science Foundation (NSF) grant (2238264). Any opinions, findings, recommendations, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems," in *Proceedings of the 2020 IEEE Symposium on Security and Privacy (S&P 2020)*, Online, May 2020, pp. 1416–1432.

[2] J. Deogirikar and A. Vidhate, "Security Attacks in IoT: A Survey," in *Proceedings of the 2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC 2017)*, Palladam, Tamil Nadu, India., Feb. 2017, pp. 32–37.

[3] M. Nawir, A. Amir, N. Yaakob, and O. B. Lynn, "Internet of Things (IoT): Taxonomy of Security Attacks," in *Proceedings of the 3rd International Conference on Electronic Design (ICED 2016)*, Sydney, Australia, Aug. 2016, pp. 321–326.

[4] Fox News, "Another Home Thermostat Found Vulnerable to Attack," https://www.foxnews.com/tech/another-home-thermostat-found-vulnerable-to-attack.amp, 2024.

[5] G. Beniamini, "Over The Air: Exploiting Broadcom's Wi-Fi Stack," https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi4.html, 2017.

[6] A. Cui, M. Costello, and S. Stolfo, "When Firmware Modifications Attack: A Case Study of Embedded Exploitation," in *Proceedings of the 2013 Network and Distributed System Security Symposium (NDSS 2013)*, San Diego, CA, Feb. 2013.

[7] D. Davidson, H. Wu, R. Jellinek, V. Singh, and T. Ristenpart, "Controlling UAVs with Sensor Input Spoofing Attacks," in *Proceedings of the 10th USENIX Workshop on Offensive Technologies (WOOT 2016)*, Austin, TX, Aug. 2016.

[8] Z. Feng, N. Guan, M. Lv, W. Liu, Q. Deng, X. Liu, and W. Yi, "Efficient Drone Hijacking Detection using Onboard Motion Sensors," in *Proceedings of the 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE 2017)*, Lausanne, Switzerland, Mar. 2017, pp. 1414–1419.

[9] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, "Securing Real-Time Microcontroller Systems through Customized Memory View Switching," in *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS 2018)*, San Diego, California, Feb. 2018.

[10] A. Khan, D. Xu, and D. J. Tian, "EC: Embedded Systems Compartmentalization via Intra-Kernel Isolation," in *Proceedings of the 2023 IEEE Symposium on Security and Privacy (S&P 2023)*, San Francisco, CA, May 2023, pp. 2990–3007.

[11] ——, "Low-Cost Privilege Separation with Compile Time Compartmentalization for Embedded Systems," in *Proceedings of the 2023 IEEE Symposium on Security and Privacy (S&P 2023)*, San Francisco, CA, May 2023, pp. 3008–3025.

[12] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "ACES: Automatic Compartments for Embedded Systems," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security 2018)*, Baltimore, MD, Aug. 2018, pp. 65–82.

[13] W. Zhou, Z. Jiang, and L. Guan, "Good Motive but Bad Design: Pitfalls in MPU Usage in Embedded Systems in the Wild," in *Black Hat Europe 2022*, London, United Kingdom, Dec. 2022.

[14] D. Cerdeira, J. Martins, N. Santos, and S. Pinto, "ReZone: Disarming TrustZone with TEE Privilege Reduction," in *Proceedings of the 31st USENIX Security Symposium (USENIX Security 2022)*, Boston, MA, Aug. 2022, pp. 2261–2279. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/cerdeira

[15] D. Kwon, J. Seo, Y. Cho, B. Lee, and Y. Paek, "PrOS: Light-Weight Privatized Secure OSes in ARM TrustZone," *IEEE Transactions on Mobile Computing*, vol. 19, no. 6, pp. 1434–1447, 2020.

[16] T. Yavuz and C. Brant, "Security Analysis of IoT Frameworks using Static Taint Analysis," in *Proceedings of the 12th ACM Conference on Data and Application Security and Privacy (CODASPY 2022)*, Baltimore, MD, Jun. 2022, pp. 203–213. [Online]. Available: https://doi.org/10.1145/3508398.3511511

[17] NXP Semiconductors, "LPCXpresso55S69 Development Board," https://www.nxp.com/design/development-boards/lpcxpresso-boards/lpcxpresso55s69-development-board:LPC55S69-EVK, 2023.

[18] Software & Systems Security Laboratory, "Public GitLab Repository for TZ-DataShield," https://gitlab.com/s3lab-code/public/tzds, 2025.

[19] A. Khan, H. Kim, B. Lee, D. Xu, A. Bianchi, and D. J. Tian, "M2MON: Building an MMIO-based Security Reference Monitor for Unmanned Vehicles," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security 2021)*, Vancouver, B.C., Canada, Aug. 2021, pp. 285–302.

[20] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer, "Protecting Bare-Metal Embedded Systems with Privilege Overlays," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P 2017)*, San Jose, CA, May 2017, pp. 289–303.

[21] J. Wang, A. Li, H. Li, C. Lu, and N. Zhang, "RT-TEE: Real-time System Availability for Cyber-physical Systems using ARM TrustZone," in *Proceedings of the 2022 IEEE Symposium on Security and Privacy (S&P 2022)*, San Francisco, CA, May 2022, pp. 352–369.

[22] N. R. Weidler, D. Brown, S. A. Mitchel, J. Anderson, J. R. Williams, A. Costley, C. Kunz, C. Wilkinson, R. Wehbe, and R. Gerdes, "Return-Oriented Programming on a Cortex-M Processor," in *Proceedings of the 2017 IEEE Trustcom/BigDataSE/ICESS (TrustCom 2017)*, Sydney, Australia, Aug. 2017, pp. 823–832.

[23] L. Luo, Y. Zhang, C. Zou, X. Shao, Z. Ling, and X. Fu, "On Runtime Software Security of TrustZone-M based IoT Devices," in *Proceedings of the 2020 IEEE Global Communications Conference (GLOBECOM 2020)*. Taipei, Taiwan: IEEE, Dec. 2020, pp. 1–7.

[24] S. Chen, J. Xu, and E. C. Sezer, "Non-Control-Data Attacks Are Realistic Threats," in *Proceedings of the 14th USENIX Security Symposium (USENIX Security 2005)*, Baltimore, MD, Jul. 2005.

[25] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks," in *Proceedings of the 2016 IEEE Symposium on Security and Privacy (S&P 2016)*, San Jose, CA, May 2016, pp. 969–986.

[26] H. Cho, J. Park, D. Kim, Z. Zhao, Y. Shoshitaishvili, A. Doupé, and G.-J. Ahn, "SmokeBomb: Effective Mitigation against Cache Side-channel Attacks on the ARM Architecture," in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services (MobiSys 2020)*, Toronto, Ontario, Canada, Jun. 2020, pp. 107–120. [Online]. Available: https://doi.org/10.1145/3386901.3388888

[27] H. Cho, P. Zhang, D. Kim, J. Park, C.-H. Lee, Z. Zhao, A. Doupé, and G.-J. Ahn, "Prime+Count: Novel Cross-world Covert Channels on ARM TrustZone," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC 2018)*, San Juan, PR, 2018, pp. 441–452. [Online]. Available: https://doi.org/10.1145/3274694.3274704

[28] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*, Barcelona, Spain, Mar. 2016, pp. 265–266.

[29] NXP Semiconductors, "MCUXpresso Software Development Kit (SDK)," https://mcuxpresso.nxp.com/en/welcome, 2023.

[30] S. Schirra, "Ropper - ROP Gadget Finder and Binary Information Tool," https://scoding.de/ropper/, 2023.

[31] K. Rubinov, L. Rosculete, T. Mitra, and A. Roychoudhury, "Automated Partitioning of Android Applications for Trusted Execution Environments," in *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*, Austin, Texas, May 2016, pp. 923–934.

[32] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares, "Virtualization on TrustZone-Enabled Microcontrollers? Voilà!" in *Proceedings of the 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2019)*, Montreal, Canada, Apr. 2019, pp. 293–304.

[33] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "CFI CaRE: Hardware-Supported Call and Return Enforcement for Commercial Microcontrollers," in *Proceedings of the 2017 International Symposium on Research in Attacks, Intrusions, and Defenses (RAID 2017)*, Atlanta, GA, Sep. 2017, pp. 259–284.

[34] J. Shi, L. Guan, W. Li, D. Zhang, P. Chen, and N. Zhang, "HARM: Hardware-Assisted Continuous Re-randomization for Microcontrollers," in *Proceedings of the 7th IEEE European Symposium on Security and Privacy (EuroS&P 2022)*, Genoa, Jun. 2022, pp. 520–536.

[35] Z. Ma, X. Tan, L. Ziarek, N. Zhang, H. Hu, and Z. Zhao, "Return-to-Non-Secure Vulnerabilities on ARM Cortex-M TrustZone: Attack and Defense," in *Proceedings of the 2023 60th ACM/IEEE Design Automation Conference (DAC 2023)*, San Francisco, CA, Jul. 2023, pp. 1–6.

[36] A. Mera, Y. Chen, R. Sun, E. Kirda, and L. Lu, "D-Box: DMA-enabled Compartmentalization for Embedded Applications," in *Proceedings of the 2022 Network and Distributed System Security Symposium 2022 (NDSS 2022)*, San Diego, California, Jan. 2022.

[37] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer, "HexType: Efficient Detection of Type Confusion Errors for C++," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, Dallas, Texas, Oct. 2017, pp. 2373–2387.

[38] D. C. Kozen, *Rice's Theorem*. Springer Berlin Heidelberg, 1977, pp. 245–248.

Table VIII: Additional evaluation results. **#Annot. SD** and **#Annot. Peri.** represents the number of annotated sensitive data and peripherals. **Identified Sensitive Data** shows the sensitive data identified by TZ-DATASHIELD's static analysis automatically. **#C SW** and **#C TRV** represent the number of compartment switches and compartments traversed in the main loop of the application. **#SFI**, **#CFI**, and **#DFI** show the total number of SFI, CFI, and DFI checks in the compartments, respectively.

	Application	#Annot. SD	#Annot. Peri.	Identified Sensitive Data	#C SW	#C TRV	#SFI	#CFI	#DFI
Bare-metal	PinLock	2	3	USART_config, rx_index	12	8	1	19	7
	Temp	2	2	ADC0_config, ADC0_commandsConfig, ADC0_triggersConfig, g_serialHandle	8	5	0	17	6
	Accel	4	2	ACCEL_I2C_config	6	3	0	9	9
	Gyro	3	3	SPI_config, g_serialHandle	12	6	0	20	10
	SD-FatFS	2	2	g_fileSystem, g_fileObject, g_sd	24	4	3	26	16
	USBVCom	1	2	g_serialHandle, g_UsbVcomConfig, s_cdcConfigList, g_UsbVcomDicEndpoints	18	4	4	21	21
FreeRTOS	PinLock	4	3	USART_config, rx_index	16	10	3	23	9
	Temp	3	2	ADC0_config, ADC0_commandsConfig, ADC0_triggersConfig, g_serialHandle	10	6	2	19	7
	Accel	5	2	ACCEL_I2C_config	8	4	1	11	10
	Gyro	4	3	SPI_config, g_serialHandle	14	7	1	22	11
	SD-FatFS	3	2	g_fileSystem, g_fileObject, g_sd	26	5	5	28	17
	USBVCom	3	2	g_serialHandle, g_UsbVcomConfig, s_cdcConfigList, g_UsbVcomDicEndpoints	22	5	5	25	23

APPENDIX A
ADDITIONAL EVALUATION

A. Developer Efforts for Annotation

Columns **Annotated Sensitive Data (#Annot. SD)** and **Annotated Peripherals (#Annot. Peri.)** in Table VIII present the developer effort required to annotate sensitive data and peripherals in the applications, respectively. Annotating sensitive data, including sensitive local, global, and heap variables is straightforward as the developer only needs to append a preprocessor provided by TZ-DATASHIELD to the variable name, as shown in Listing 2. Annotating peripheral data requires the developer to specify the base address and size of the peripheral MMIO region to be protected using a preprocessor. This information can be found in the MCU's reference manual or SDK header files, which proficient developers can easily identify. Column **Identified Sensitive Data** shows the number of other sensitive data points identified by TZ-DATASHIELD's static analysis tool. These data points are not annotated by developers but are automatically identified by the tool. Both the annotated and identified sensitive data are used to generate the compartmentalization configuration.

B. Compartment Switches and Traversing

Columns **Compartment Switches (#C SW)** and **Compartment Traversed (#C TRV)** in Table VIII show the number of compartment switches and compartments traversed in the main loop of the applications, respectively. These numbers provide more insights into the runtime overhead of TZ-DATASHIELD. A typical MCU application consists of the initialization and the main loop phases. The initialization phase configures the MCU peripherals while the main loop handles the core application logic. The reported numbers of compartment switches and traversals are measured using the compartments traversed in the main loop only as these are repeatedly executed. We exclude compartments used in the initialization phase as they are invoked only once.

C. Accuracy of Static Analysis

This metric measures the accuracy rates of over-approximation and under-approximation. The accuracy of the static analysis tool in identifying sensitive data flow units is crucial for the effectiveness of TZ-DATASHIELD. *Over-approximation* occurs when the analysis identifies data flow that does not exist in reality, leading to potential false positives. While conservatively isolating more code and data in a compartment with over-approximation does not reveal sensitive data to potential attacks, it may increase the performance overhead of TZ-DATASHIELD. *Under-approximation* involves missing data flow, resulting in false negatives. This poses a security risk, as untracked sensitive data flows could be exploited by an adversary, undermining the protecting mechanisms enforced by TZ-DATASHIELD. Our results show that TZ-DATASHIELD identified 16 actual data flows out of 18 reporting ones due to our conservative configuration with Andersen's points-to algorithm. While this accuracy is aligned with industry-leading tools, it is worth noting that the inherent limitations posed by Rice's Theorem [38] prevent the absolute identification of all data flows.

D. Case Studies

To demonstrate the protection provided by TZ-DATASHIELD, we selected three applications (Accel, SD-FatFS, and USBVCom) and examined ways in which an attacker can access sensitive data. These applications cover several commonly used peripherals with different operation complexity.

1) Accel: This application reads an accelerometer sensor through the I2C bus and sends data through the USART port.
Security Objective. We want to protect the integrity of the acceleration data, preventing an attacker from modifying the data.
SDF Protection. The peripheral I2C and USART are assigned to the secure world, ensuring that TrustZone prevents any code in the normal world from accessing these peripherals. This isolation safeguards against attacks originating from corruption in normal-world firmware. Moreover, corrupted

compartments cannot bypass integrity checks performed by the security monitor before being loaded. Even if a compromised compartment were to be loaded and executed, the SFI/CFI/DFI mechanisms enforce constraints on destination addresses. For instance, a compartment responsible for initializing the I2C bus is restricted from accessing the I2C data register, while a compartment operating the sensor via the I2C bus is prohibited from modifying configuration-related registers.

2) SD-FatFS: This application reads/writes data on an SD card.

Security Objective. We want to keep the confidentiality and integrity of the data from other parts of the application.

SDF Protection. Given the diverse sources of data written to or read from the SDIO, it is critical to protect both global and local variables that may ultimately influence or be influenced by the data on the SD card. To achieve this, all functions involving such variables must be placed within a compartment. An attacker could attempt to steal or modify the data through two potential methods: indirectly accessing the aforementioned variables or directly accessing the SDIO's data register. However, as discussed in previous scenarios, both approaches are rendered infeasible due to the implemented security mechanisms.

3) USBVCom: This application emulates a virtual COM port or serial port. The COM port is widely used in embedded systems and devices, such as medical equipment, GNSS modules, Bluetooth, and WiFi modules. A USB virtual COM port is typically used to connect such devices to a desktop or server computer to transfer data.

Security Objective. We want to defend our data transferred using this interface against stealing or tampering attempts.

SDF Protection. To achieve this, the USB port and the USART used by this application need to be assigned to the secure world, and the code used to operate USART and USB needs to be put into compartments. There are three ways in which an attacker could steal or tamper with data during transfer: **(1)** directly accessing the USART data register connected to the device, **(2)** accessing the USB's data buffer, and **(3)** reading or writing local variables used to transfer data from the USB buffer to the USART data register. With TZ-DATASHIELD, the USART data register, the USB buffer, and the stack containing local variables are all protected by assigning the register to the secure world and protecting the buffer and the stack. Additionally, the SFI/CFI/DFI mechanisms ensure that `load` instructions cannot transfer data to the normal world, as they verify the destination address of each `load` instruction, preventing attackers from exploiting compartment code.

E. Formal Verification

To formally verify the correctness of SFI, CFI, and DFI enforcement of TZ-DATASHIELD using CBMC, we ported the essential functions that ensure the enforcement of these mechanisms to a desktop machine running Linux. We also model the SFI, CFI, and DFI checks as assertions in the test code, and then use CBMC to check the correctness of these assertions. The results show that the SFI, CFI, and DFI checks are correctly enforced in the firmware code.

1) SFI: To verify the correctness of the SFI enforcement, we need to ensure the correctness of `check()` function, which can be modeled as:

$$\begin{aligned} check(\text{addr}, \text{i}) = true &\Rightarrow s_i \leqslant \text{addr} \leqslant e_i, \\ check(\text{addr}, \text{i}) = false &\Rightarrow \text{addr} < s_i \vee \text{addr} > e_i, \end{aligned} \tag{1}$$

where s_i and e_i are the start and end addresses of the compartment identified by the ID i, respectively.

2) CFI and DFI: To verify the correctness of the CFI enforcement, we need to ensure the correctness of shadow stack operation functions (`shadow_push()` and `shadow_pop()`). The following equations show the model of shadow stack operation functions:

$$\begin{aligned} \text{shadow\_push}(\text{addr}) \Rightarrow S.top &\leftarrow prev\_top + 4 \\ &\wedge S.data[S.top] = v, \\ \text{shadow\_pop}(\text{addr}) \Rightarrow result &= S.data[S.top] \\ &\wedge S.top \leftarrow prev\_top - 4, \end{aligned} \tag{2}$$

where S is the shadow stack, $S.top$ is the top of the shadow stack, $S.data$ is the data stored in the shadow stack, and v is the value to be pushed into the shadow stack.

3) DFI: To verify the correctness of the DFI enforcement, we need to ensure the correctness of the `update_rdt()`, `get_rdt()` and `check_store()`, which can be modeled as:

$$\begin{aligned} \text{update\_rdt}(\text{addr}, \text{store\_id}) &\Rightarrow \\ \text{RDT}[\text{addr}] &\leftarrow \text{store\_id}, \\ \text{get\_rdt}(\text{addr}) &\Rightarrow \\ result &= \text{RDT}[\text{addr}], \\ \text{check\_store}(\text{addr}, \text{store\_id}) = true &\Rightarrow \\ \text{RDT}[\text{addr}] &\in Allowed\_id, \\ \text{check\_store}(\text{addr}, \text{store\_id}) = false &\Rightarrow \\ \text{RDT}[\text{addr}] &\notin Allowed\_id, \end{aligned} \tag{3}$$

where RDT is the RDT, and Allowed_id is the set of `store_id` that are allowed to store data to the address addr according to the static analysis result.

APPENDIX B
ARTIFACT APPENDIX

TZ-DATASHIELD is a novel LLVM compiler tool that enhances ARM TrustZone with sensitive data flow (SDF) compartmentalization, offering robust protection against strong adversaries in MCU-based systems. This artifact provides detailed information on how to access the source code of TZ-DATASHIELD and how to run and evaluate it.

A. Description and Requirements

This section lists all the information necessary to recreate the experimental setup to run the artifact.

1) How to Access: The artifact is available at a public repository [18], which contains the source code referenced in this paper. We provided step-by-step instructions to run the artifact in the `README.md` file in the repository. The artifact is also available at https://doi.org/10.5281/zenodo.14257983.

2) Hardware Dependencies: Our artifact requires a commodity desktop machine with an x86-64 CPU and an ARMv8-M microcontroller unit (MCU) with TrustZone-M support. Specifically, we used an LPCXpresso55S69 Development Board* from NXP Semiconductors, which is equipped with an ARM Cortex-M33 processor.

3) Software Dependencies: The artifact requires a Linux-based operating system. We tested the artifact on Ubuntu 22.04 LTS.

4) Benchmarks: None.

B. Artifact Installation and Configuration

We first require that our repository is cloned to a local directory via `git clone`, and then follow the instructions from `README.md` file in the repository.

C. Experiment Workflow

There are **four** main steps to run the experiments: **(1)** Build the compiler tool of TZ-DATASHIELD. TZ-DATASHIELD utilizes two compilers, GCC 11.4.0 and Clang/LLVM 14.0.6.The clang compiler is used to compile the source code of the MCU application to object files (`.o`) and run LLVM passes to enforce SFI, CFI, and DFI. The GCC compiler links the object files into two different firmware images: one for the normal world and one for the secure world. **(2)** Run static program analysis. Building the static analysis tool of TZ-DATASHIELD requires a general-purpose compiler on the host machine. **(3)** Build the firmware images. The firmware images are compiled using the compilers that we just built. Use the `Makefile` provided in the repository to compile the firmware. **(4)** Download the firmware images to the MCU. The MCU development board has an on-board J-Link debugger that can be used to download the firmware images to the MCU. The download process can also be automated using the `Makefile`.

*https://www.nxp.com/design/design-center/software/development-software/mcuxpresso-software-and-tools-/lpcxpresso-boards/lpcxpresso55s69-development-board:LPC55S69-EVK

D. Major Claims

- **Claim 1:** The static analysis tool of TZ-DATASHIELD can automatically identify sensitive data flows in the annotated source code and output the function/global variable list that needs to be compartmentalized.
- **Claim 2:** The compiler tool of TZ-DATASHIELD can generate compartmentalized firmware images with SFI, CFI, and DFI enforcement.

E. Evaluation

Before starting the evaluation, ensure that the static analysis tool and the compilers for TZ-DATASHIELD are successfully built and that the MCU development board is connected to the host machine. To verify the connection, run the `lsusb` command in the terminal, and the output should display the J-Link debugger, as shown.

```
TZDS> lsusb
...
Bus 001 Device 003: ID 1366:1024 SEGGER J-Link
...
```

For detailed steps, follow the instructions in the `README.md` file in the repository.

1) Experiment (E1): [Sensitive Data Flow Identification] [3 human-minutes + 10 compute-minutes]: This experiment aims to identify the sensitive data flows in the annotated source code.

[Preparation] Open a new terminal and navigate to one application directory, for example, `adc` (there are a total of 12 applications that can be tested with).

[Execution] Run the static analysis tool of TZ-DATASHIELD by invoking commands:

```
cd lpc_firmware
# use adc as an example
cd adc
# install/update template code (including SDK)
./install-template.sh
make clean && make COMPILER=clang -j $(nproc)
make sdf
```

[Results] The `.yaml` file shows the compartments, peripheral, and shared global variables.

```
$ cat output/comp.yaml
base:
  DbgConsole_Init, FLEXCOMM_GetInstance, ...:
    - g_serialHandle
    - USART
  DbgConsole_Printf, ...:
    - g_SensorData, g_serialHandle
    - USART
  LPADC_Init, LPADC_DoResetConfig, ...:
    - ADC0_config
    - ADC
  ...
```

2) Experiment (E2): [Compartmentalized Firmware Generation] [5 human-minutes + 20 compute-minutes]: This experiment aims to generate the compartmentalized firmware images with SFI, CFI, and DFI enforcement.

[Preparation] Same as E1.

[Execution] Run the compiler tool of TZ-DATASHIELD by invoking commands:

```
cd lpc_firmware
# use adc as an example
cd adc
# install/update template code (including SDK)
./install-template.sh
make clean && make COMPILER=clang -j $(nproc)
ls -l output/adc.hex
```

[Results] The compiler tool will generate the firmware images under `output` directory.

3) Experiment (E3): [Download Firmware Images to MCU] [10 human-minutes]: This experiment aims to download the firmware images to the MCU and test if the firmware functions as expected.

[Preparation] The first step is the same as E1. Then, open another terminal and run `minicom -c on -b 115200 -D /dev/ttyACM0` to open a serial terminal connected to the MCU.

[Execution] Download the firmware images to the MCU by invoking commands:

Terminal 1:

```
cd lpc_firmware
# use adc as an example
cd adc
./install-template.sh
make clean && make COMPILER=clang -j $(nproc)
make download
```

Terminal 2:

```
$ minicom -c on -b 115200 -D /dev/ttyACM0

Welcome to minicom 2.8

OPTIONS: I18n
Port /dev/ttyACM0, 11:42:50

Press CTRL-A Z for help on special keys

Current temperature:  28.44
Current temperature:  28.47
...
```

[Results] If the firmware runs successfully, the terminal will display the output of the tested application in terminal 2.

F. Artifact Changes for Minor Revision

We provide instructions to run the following additional experiments that we provide beyond the initially submitted paper. Specifically, our shepherd has requested to "add statistics about analyses of applications," which we address with the following experiments:

1) Experiment (E4): [Over-approximation and Under-approximation Rate] [10 human-minutes + compute-minutes]: This experiment aims to measure the accuracy of TZ-DATASHIELD's static analysis tool by using well-understood C/C++ programs with known data slices.

[Preparation] Open a new terminal and navigate to `slicing_benchmark`.

[Execution] Run the static analysis tool by invoking commands: *[Results]* The actual and analyzed program slices will

```
cd slicing_benchmark/1
./compile.sh
```

show on the terminal with the last line showing the rates.

```
$ ls
1.c               1.svf.bc  compile.sh
groundtruth.png   icfg.dot  vfg.dot
1.ll              callgraph.dot
full_svfg.dot     groundtruth.svg
pag.dot           vfg_model.dot
```

Now you can compare the ground truth value flow graph (`groundtruth.svg`) and the generated value flow graph (`vfg_model.dot`, using an online tool, like https://dreampuf.github.io/).

2) Experiment (E5): [SFI, CFI, and DFI Enforcement] This experiment aims to check if the SFI, CFI, and DFI mechanisms can block illegal access to global variables and peripherals. The attacks are implemented as maliciously modified compartments.

[Preparation] Open a new terminal and navigate to `attacks/<attack number>`. Then, open another terminal and run `minicom -c on -b 115200 -D /dev/ttyACM0` to open a serial terminal connected to the MCU.

[Execution] There are four different attacks, launch the attacks by invoking commands:

Terminal 1:

```
cd attacks/1
# or cd attacks/2
make download
```

Terminal 2:

```
$ minicom -c on -b 115200 -D /dev/ttyACM0

Welcome to minicom 2.8

OPTIONS: I18n
Port /dev/ttyACM0

Press CTRL-A Z for help on special keys

SFI violation detected!
```

[Results] The terminal connected to the MCU development board should show an illegal access detected message.