OPEC: Operation-based Security Isolation for Bare-metal Embedded Systems

Anonymous Author(s)
Submission Id: 185

Abstract

Bare-metal embedded systems usually lack security isolation. Attackers can subvert the whole system with a single vulnerability. Previous research intends to enforce both privilege isolation (to run application code at the unprivileged level) and resource isolation for global variables and peripherals. However, it suffers from partition-time and execution-time over-privilege issues, due to the limited hardware resources (MPU regions) and the improper way to partition a program.

In this paper, we propose operation-based isolation for bare-metal embedded systems. An *operation* is a logically independent task composed of an entry function and all functions reachable from it. To solve the partition-time over-privilege issue, we utilize the global variables shadowing technique to reduce the needed MPU regions to confine the access of the global variables. To mitigate the execution-time over-privilege issue, we split programs into code compartments (called operation) that only contain necessary functions to perform specific tasks, thereby removing the resources needed by unnecessary functions. We implement a prototype called OPEC, which contains an LLVM-based compiler and a reference monitor. The compiler partitions a program and analyzes the resource dependency for each operation. With the hardware-supported privilege levels and MPU, the reference monitor is responsible for enforcing the privilege and resource isolation at runtime. Our evaluation shows that OPEC can achieve the security guarantees for the privilege and resource isolation with negligible runtime overhead (average 0.23%), moderate Flash overhead (average 1.79%), and acceptable SRAM overhead (average 5.35%).

1 Introduction

Bare-metal embedded systems have surged rapidly in recent years. They run a monolithic software that provides both system functionality and application logic without the intervention of the operating system. Bare-metal embedded systems are vulnerable due to the absence of security defenses, such as privilege isolation, data execution prevention (DEP), and address space layout randomization (ASLR) [12]. Those defenses are widely deployed in desktop systems, but they cannot be directly adopted due to insufficient hardware resources and limited hardware security extensions. For instance, desktop systems implement resource isolation with the hardware called Memory Management Unit (MMU). However, bare-metal embedded systems have neither MMU nor sufficient resources to support such isolation. Furthermore, there is no isolation between tasks, i.e., different execution stages of a program.

Nevertheless, the attack techniques targeting desktop systems can be easily transferred to bare-metal embedded systems. Attackers can get full control of the whole system by exploiting a vulnerability in any task. Hence, enforcing secure isolation on bare-metal embedded systems is an effective way to secure the devices.

Previous research provides security isolation to bare-metal embedded systems. However, it either misses or has defects in resource isolation. EPOXY [19] provides privilege isolation by identifying and executing only sensitive instructions at the privileged level. Still, EPOXY does not support flexible resource isolation between tasks. MINION [25] implements privilege and thread-level resource isolation for real-time microcontroller systems. MINION predefines the accessible memory range and permissions for each thread during the program compartmentalization. Nevertheless, the accessible memory is oversized, and its permission is overset due to hardware constraints, i.e., the limited number of MPU regions. Additionally, threads are absent in many bare-metal embedded systems, so MINION cannot be applied to bare-metal embedded systems directly.

To enforce privilege isolation and more fine-grained resource isolation, ACES [18] adopts a code-module based strategy that partitions a program based on the source files or peripherals. However, ACES suffers from two over-privilege issues. First, ACES allows a compartment to access unnecessary global variables. For a specific compartment, different parts of its accessible global variables may be shared with related compartments. Due to the limited number of MPU regions, ACES directly grants all the related compartments to all of these global variables (called partition-time over-privilege issue in this paper). Second, ACES partitions a program without considering the control flow. ACES involves unnecessary functions into a compartment to complete a specific task, e.g., unlocking a smart lock. Therefore, a compartment can access unnecessary resources at runtime (called execution-time over-privilege issue in this paper). Furthermore, ACES limits the number of accessible peripherals of a compartment due to the limited number of MPU regions, which brings inflexibility when partitioning a program.

In this paper, we propose operation-based security isolation. It provides both privilege isolation and resource isolation to bare-metal embedded systems. *An operation is a*

```c
void Unlock_Task() {
  HAL_UART_Receive_IT(&PinRxBuffer);/*buggy*/
  result = hash(PinRxBuffer);
  if(compare(result, KEY))
    do_unlock();
}

void Lock_Task() {
  HAL_UART_Receive_IT(&PinRxBuffer);/*buggy*/
  if(PinRxBuffer[0]=='0')
    do_lock();
}

int main(void) {
  System_Init();/*Task1: configure core peripherals*/
  Uart_Init();  /*Task2: configure UART*/
  Key_Init();   /*Task3: compute the hash of PIN,
                  and save it to KEY*/
  Init_Lock();  /*Task4: Init lock*/
  while (1)
  {
    Unlock_Task(); /*Task5: perform the unlock task*/
    Lock_Task();   /*Task6: perform the lock task*/
  }
}
```

Listing 1. Main logic of the application PinLock.

logically independent task composed of an entry function and all functions reachable by that entry function. For instance, as shown in Figure 1, the PinLock application can be split to six tasks (operations). As in Listing 1, when PinLock receives the correct pin code, the operation `Unlock_Task` invokes the function `do_unlock` to perform unlocking. When executing inside one operation, the code can only access necessary resources, i.e., the global data and peripherals needed to complete the task. The code cannot access other resources. This method provides the security guarantee that a vulnerable task cannot compromise other ones or the whole system.

For the partition-time over-privilege issue, we solve it through the *global data shadowing* technique. We make a shadow copy of each shared global variable of one operation and put it into a continuous memory space that is only accessible by that operation. By doing so, the continuous memory space only contains global data necessary for a specific operation, which avoids incorporating redundant resources. For the execution-time over-privilege issue, we mitigate it through an approach that one operation only includes the functions needed to perform a specific task. Indeed, a function may contain some basic blocks that are not executed at runtime, which could also cause the execution-time over-privilege issue. Furthermore, our system virtualizes the MPU regions to use the MPU flexibly and facilitate the operation partitioning. Explicitly, our system supports reconfiguring the MPU on demand at runtime.

We implement a prototype called OPEC. It consists of two components, i.e., `OPEC-Compiler` and `OPEC-Monitor`. The former is an LLVM-based compiler that is responsible for partitioning the program into different operations and identifying needed resources for each operation. The latter is responsible for global data shadowing, stack isolation, and operation switching. Specifically, `OPEC-Compiler` gets the application source code and the operation entry function list specified by developers. Then it performs a static analysis to identify

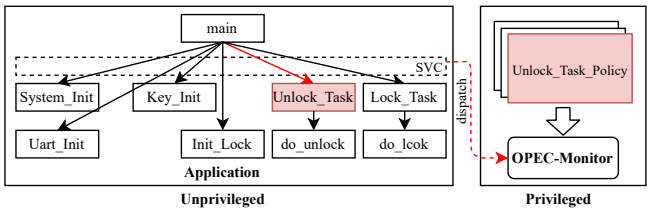

Figure 1. Example of the operation partitioning, switching, and runtime enforcement to PinLock. The PinLock can be divided into six tasks, each task can be regarded as an operation. At runtime, `SVC` instructions trigger the operation switch, and the `OPEC-Monitor` of OPEC performs the switch and updates the MPU configurations according to the policy to enforce the security isolation.

the global data and peripherals needed by each operation to generate the *operation policy*. After that, it uses the *operation policy* to produce the final program image. `OPEC-Monitor` is linked to the image during the compiling time to enforce the isolation policy. The operation switch is triggered by a software interrupt (the `SVC` instruction), which will be handled by `OPEC-Monitor`. `OPEC-Monitor` first performs data synchronization based on the corresponding policy. Then, it enforces resource access permissions by setting up the MPU.

We evaluate the effectiveness of OPEC with six representative applications and the CoreMark benchmark [4]. The evaluation shows that our system effectively enforces privilege isolation and reduces the resources that can be accessed for an operation at runtime. Compared to the vanilla system, an average of 37.79% of the resources are accessible at runtime. We further evaluate the performance overhead. The average runtime overhead is 0.23%. OPEC consumes low Flash and moderate SRAM resources, i.e., 1.79% and 5.35% for Flash and SRAM, respectively.

In summary, the contributions of our work are as follows:

- We analyze the over-privilege issues of the previous security isolation for bare-embedded systems, which motivates our work (Section 3).
- We propose operation-based security isolation and solves or mitigates the over-privilege issues (Section 4 and Section 5).
- We prototype our system OPEC and evaluate it with six representative applications on two different development boards. The experiment shows that OPEC enforces privilege isolation and resource isolation for bare-metal embedded systems with a negligible impact on performance. (Section 6).

2 Background

2.1 Memory Address Space and Privilege Levels

OPEC works towards bare-metal embedded systems [19], whose system libraries and application code are statically linked and executed on low-end microcontrollers, e.g., ARM Cortex-M CPU families. In this paper, we use the popular

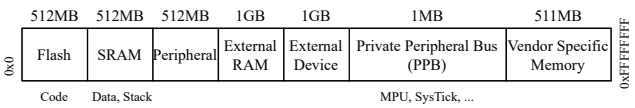

Figure 2. Memory layout of the ARMv7-M architecture.

ARMv7-M architecture [2] as a reference to describe our system. This choice complies with previous work [18, 19].

The address space of the ARMv7-M architecture contains code, data, peripherals, and so forth., as shown in Figure 2. For instance, the code usually resides in the lower 512 MB address space, ranging from `0x00000000` to `0x1FFFFFFF`. The program data, including the stack and global variables, are in the SRAM region. For peripherals, they are either at the peripheral region or the external device region. Despite a large address space, only a small portion is used in a real system. A bare-metal embedded system usually has a few MBs of Flash and KBs of SRAM.

The ARMv7-M architecture has two privilege levels for software execution, i.e., privileged and unprivileged levels. The program can execute some security-sensitive instructions at the privileged level, such as configuring the hardware directly. The unprivileged level is reserved for the application code. The unprivileged application can execute a `SVC` instruction to make a supervisor call to escalate to the privileged level. Although the microcontroller supports privilege isolation, it is not fully leveraged by bare-metal embedded systems. Both system libraries and applications are running at the *privileged level*. Our system intends to provide privilege isolation. Note that only privileged software is allowed to access memory range within Private Peripheral Bus (PPB), where core peripherals such as MPU and SysTick timer reside. Unprivileged access will trigger a bus fault.

2.2 Memory Protection Unit (MPU)

Memory Protection Unit (MPU) is used to provide physical memory access control. It defines several regions by specifying the base address, size, and access permissions at the privileged and unprivileged levels. Each region is labeled with a number. If two memory regions have an overlapped memory range, the access to this range is confined by the region labeled with the highest number. Accessing the memory range prohibited by the MPU will trigger a memory management fault. However, there are some restrictions on the region configuration. The region size must be the power of two, and the smallest permitted region size is 32 bytes. The region's base address must be aligned with its size. Therefore, it is challenging to enforce precise access control for a large number of scattered address spaces by the limited number of MPU regions. One MPU region can be further split into eight equal-sized sub-regions, and each can be enabled or disabled individually. If a sub-region of a region is disabled, a lower-numbered MPU region that overlaps this memory range confines the memory access to it.

3 Overall Design and Threat Model

3.1 Motivation

Our system intends to provide security isolation to bare-metal embedded systems. By doing so, one compromised task cannot access arbitrary resources (data and peripherals). The data may contain security-critical system states. Peripherals, as the interface between the device and the outside world, may directly impact the physical world. For instance, by writing a special peripheral register, the attacker can control a robot arm's movement speed causing safety issues.

Though the previous system [18] intends to provide security isolation, it suffers from two over-privilege issues. We use two examples derived from the PinLock application to illustrate two different types of the over-privilege issues.

Partition-time Over-privilege results from the limitation of MPU. Figure 3 shows the partition-time over-privilege issue when using the MPU to isolate global variables. ACES solves the over-privilege caused by the size and alignment restriction of the MPU region by rearranging the addresses of global variables. However, it fails to solve the over-privilege issue caused by limited MPU regions. Different compartments may share variables, and most of the shared variables would become a separate region. It leads to exceeding the available MPU regions when a compartment has too many shared variables among different compartments. Therefore, ACES merges some regions to solve this issue at the cost of introducing the over-privilege issue. In Figure 3(a), compartment `C2` groups `V1` and `V3` to save the use of regions. It leads to the over-privilege issue for compartment `C3` since `C3` only needs to access V1.

Our system solves this problem through the global data shadowing technique that each compartment has a shadow copy of the shared global variables (Figure 3(b)). These variables are rearranged inside a continuous memory range that is confined by one MPU region.

Execution-time Over-privilege is caused by including unnecessary code when executing a task. As shown in Figure 4, the compartment `C1` contains multiple functions. However, when performing a task (e.g., unlocking a smart lock), only the function `F1` in the compartment `C1` executes. Resources that are accessed by other functions in compartment `C1` become redundant. If the code in compartment `C1` is compromised, it can access these redundant resources, which leads to execution-time over-privilege.

Moreover, the code-module based way to divide compartments [18] also induces frequent compartment switches because the execution flow of a specific task may cross multiple compartments. For instance, when performing task two in Figure 4, five compartment switches between compartments are needed. Our system considers the execution flow when partitioning a program. It mitigates the execution-time over-privilege issue while reducing the overhead of compartment switches at the same time.

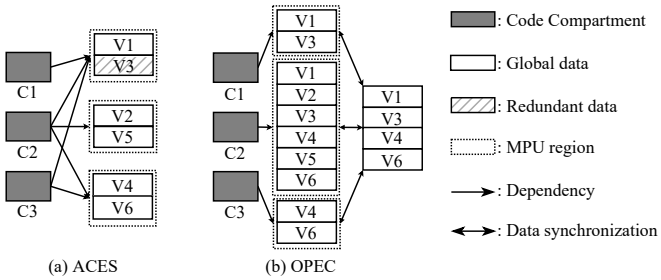

Figure 3. Example of the partition-time over-privilege issue derived from the PinLock.

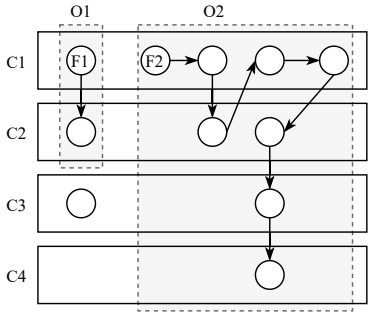

Figure 4. Example of the execution-time over-privilege issue derived from the PinLock.

3.2 System Overview

The workflow of our system is shown in Figure 5. It consists of two key stages, *compiler-assisted operation partition* and *hardware-assisted operation isolation*. These two stages are involved by the following two components, i.e., `OPEC-Compiler` and `OPEC-Monitor`, respectively. The former is an LLVM-based compiler, which analyzes the program and builds the image with the operation isolation property. The latter enforces privilege and resource isolation between operations at runtime. It is linked to the application code during compiling.

3.3 Assumptions and Security Benefits

Our system assumes a strong threat model that the attacker can gain the arbitrary code execution capability to construct the primitives to read from and write to arbitrary memory locations. These primitives could be achieved through hijacking the control flow of the program, e.g., corrupting function pointers, and then by leveraging Return-Oriented Programming (ROP) to construct powerful attack primitives. However, attackers cannot inject code since the writable memory regions are not executable.

We assume that the hardware has two privilege levels (privileged and unprivileged) and a memory protection unit (MPU). In addition, we assume that the system is a bare-metal one whose program image is a statically linked binary. It does not support dynamically linked libraries. It is consistent with the bare-metal embedded systems in the real world. These assumptions comply with the previous system [18].

OPEC enforces the least-privilege principle to bare-metal embedded systems and provides the following protections. First, it ensures the isolation between operations. The compromise of one operation cannot directly take over the whole system. Second, it enforces the least-privilege principle of operation. The compromise of one operation can only manipulate the resources that are accessible in that operation. Third, the sanitization of changes to global data *alleviates* the cross-domain threat that corrupts safety-critical global variables [1]. Our system reduces the security consequences of bare-metal embedded systems after being compromised, providing a practical defense with a powerful threat model.

4 Compiler-assisted Operation Partitioning

This section presents how `OPEC-Compiler` partitions operations and generates the program image with our isolation property. Our system takes the program source code and the list of operation entry functions as inputs to produce the final image (Figure 5). In our prototype, the `OPEC-Compiler` is built on LLVM 9.0 [11], with new passes to perform the program analysis and instrumentation.

4.1 Call Graph Generation

An operation is a specific task of the bare-metal system. From the program's perspective, *an operation is a subtree in the call graph, with a developer-provided function as the root.* With the call graph, we can easily traverse all the functions from the root node. However, building a precise call graph is challenging due to the indirect function calls (*icalls* for short) [31]. `OPEC-Compiler` leverages the Andersen point-to analysis implemented by SVF [34] to determine the potential legal targets of function pointers. Nevertheless, there are some icalls that cannot be resolved by SVF. In this case, we use type-based analysis to find the potential targets. We consider two function types identical if the number of arguments, the type of the structure argument, the type of the pointer argument, and the type of the return value are the same. The indirect function call edges are then added to the call graph to ensure its soundness. Note that the results of the point-to analysis are conservative and over-approximated, which contains false positives. Otherwise, an unsound call graph will bring dependency miss to operations.

4.2 Resource Dependency Analysis

After constructing the call graph, the next step is to analyze the needed resources of each function. The resources include global variables and peripherals. `OPEC-Compiler` leverages state-of-the-art static analysis techniques to perform a conservative analysis.

Global Variables `OPEC-Compiler` leverages the forward slicing to identify accessed global variables in each function.

[1]Note that if attackers can directly manipulate safety-critical peripherals that are accessible in a compromised operation, the sanitization cannot prevent this attack.

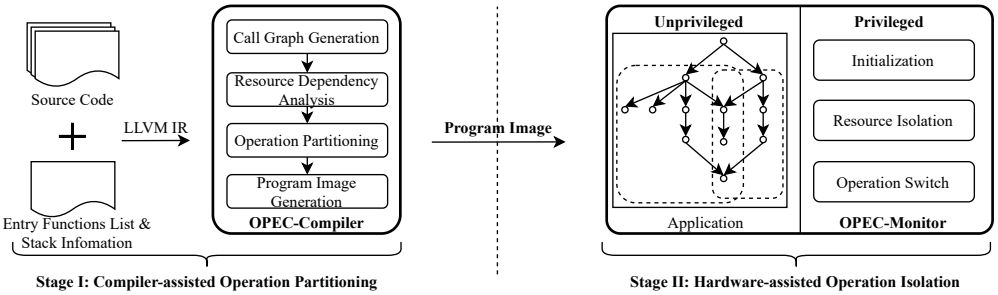

Figure 5. The workflow of OPEC.

Global variables can be directly and indirectly accessed and our system processes them differently.

Direct accesses can be identified by recognizing all the load and store instructions that use the global variables as operands. `OPEC-Compiler` uses the built-in def-use analysis of the LLVM to identify. Indirect access happens when a variable is accessed through a pointer. `OPEC-Compiler` performs an inter-procedural point-to analysis using the state-of-the-art tool SVF [34]. The analysis identifies local and global variables targets that a pointer may point to. We further filter out the local targets and leave the global ones.

`OPEC-Compiler` also records the pointer fields of a global variable by leveraging its type. This information is further used for updating pointer fields of shadow variables when switching an operation (Section 5.3).

Peripherals are accessed through memory-mapped addresses (Figure 2) in bare-metal systems. For a specific System-on-Chip (SoC), memory addresses of peripherals are constant. `OPEC-Compiler` uses this information to identify peripheral access. We obtain the addresses of peripherals from the SoC datasheet. `OPEC-Compiler` uses the backward slicing at IR level to examine whether the operand of a load/store instruction contains a constant memory address. It then compares the address with a predefined list of peripheral addresses retrieved from the datasheet to check whether the address is peripheral. If so, the `OPEC-Compiler` marks it as a peripheral accessible by the function where the `store/load` the instruction belongs to. Furthermore, we divide the accessible peripherals into two lists for general peripherals and core peripherals, since accessing core peripherals (on PPB) requires the code to run at the privileged level (Section 2).

4.3 Operation Partitioning

This step aims to partition a program into operations and identify all resources needed by each operation. `OPEC-Compiler` depends on the entry functions list provided by the developers to partition a program (Figure 5). The process of selecting the entry function of an operation is straightforward. Take the application PinLock as an example (Listing 1). This application can be divided into six tasks. We can treat each task as an operation and choose the root node of each task in the call graph as the entry function. Apart from the operations generated through the entry function list, `OPEC-Compiler` also considers the function `main` as a default operation. The default operation can access the resources necessary to the function `main`. Note that the operation entries cannot be functions within an interrupt handling routine.

For each operation entry function, `OPEC-Compiler` performs the Depth-First-Search (DFS) algorithm to traverse the call graph from the entry function to determine the functions that operation contains. Note that our customized DFS algorithm will perform backtracking when it reaches the entry function of another operation.

After that, the `OPEC-Compiler` merges all the resources needed by the member functions of each operation to produce the resource dependency. In our design, each peripheral is protected by an individual MPU region. To save the use of the limited MPU regions, `OPEC-Compiler` will first sort the peripherals needed by one operation at the ascending order of their start addresses. Then the adjacent peripherals will be merged and protected by an individual MPU region. Even so, one operation may need regions exceeding the limitation of the MPU to access the peripherals. In this scenario, OPEC *virtualizes* the usage of MPU regions to solve this issue. The details are in Section 5.2. Alternatively, the developers can choose to split one large operation into smaller ones. Since smaller operations may access fewer peripherals and need fewer MPU regions. Finally, `OPEC-Compiler` generates a policy file that contains accessible resources of each operation.

4.4 Program Image Generation

After partitioning the program, the `OPEC-Compiler` generates the operation data sections and MPU configurations, then instruments the program to produce the final image.

Operation Data Section OPEC uses the global data shadowing technique to isolate global variables. Specifically, each operation has an exclusive *operation data section* that contains all the global variables it needs. An operation can access global variables only within its operation data section. Each operation data section is confined by one MPU region preventing variables from being corrupted by other operations.

Our system divides the global variables into two types, *internal* and *external*. The internal ones are accessed by only one operation and directly placed into the corresponding operation data section. The external ones are accessed by two or more operations. Each external variable has shadow

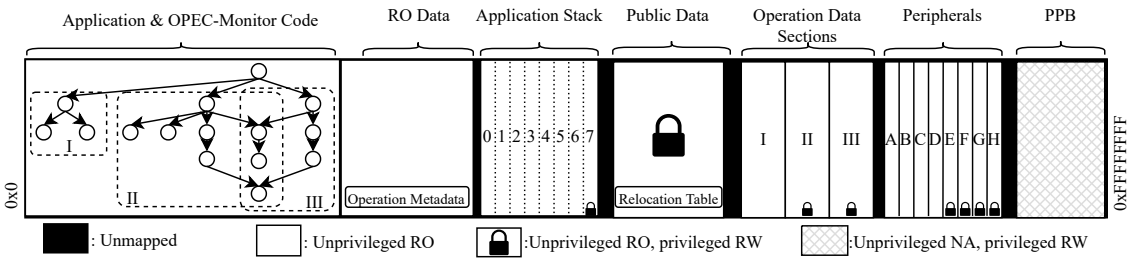

Figure 6. The memory layout of the program armed with OPEC. For operation I, it can modify the stack except for the sub-region 7 of the stack used by the previous operation, its operation data section, and four peripherals. RO: Read-only; RW: Read and write; NA: No access.

copies in corresponding data sections. Those shadow copies are accessed through a *variables relocation table*. Specifically, the `OPEC-Compiler` creates a data pointer for each external variable at the variables relocation table. The data pointer either points to the original variable or the shadow copy. The details of using the relocation table for data accessing will be discussed in Section 5.2.

In our design, each operation data section is protected by one MPU region. However, the MPU has a rigid address alignment requirement (Section 2). Improper placement of operation data sections will cause external memory fragments. To reduce the external fragments, the `OPEC-Compiler` first sorts the operation data sections according to their sizes in descending order. Then it calculates the start addresses of these sections and places them accordingly.

Operation Metadata The metadata of each operation contains five parts, i.e., MPU configurations, stack information, sanitization value of critical global variables, a peripheral list, and a variable relocation table. The MPU configurations, which are generated by the `OPEC-Compiler` after the building of operation data sections, are used to confine the memory access permissions of each operation. The stack information and sanitization value are provided by the developers. The former is used to annotate the pointers in the entry function arguments, which facilitates the `OPEC-Monitor` performing stack synchronization during the operation switching. The latter is used for global variables sanitization. The peripheral list is used as an allow list when accessing the peripherals.

Code Instrumentation `OPEC-Compiler` inserts the initialization routines before entering the function `main` of the application. We will illustrate the initialization in Section 5.1. Moreover, `OPEC-Compiler` inserts `SVC` instructions before and after the call site of each operation entry function. The application code will escalate to the privileged level after executing the `SVC` instruction. After that, the control flow will be transferred to the operation switch routine. When the program enters into or exits from an operation, the operation switch is invoked to set the new execution environment or recover from the previous execution context. After instrumentation, the library that contains the `OPEC-Monitor` is linked to the application code to produce the final image.

5 Hardware-assisted Operation Isolation

This section illustrates the `OPEC-Monitor` of our system. It enforces privilege isolation, resource isolation, and operating switching based on the policy generated in the previous stage.

5.1 Initialization

`OPEC-Monitor` initializes the system before running the application code. First, it initializes the operation data section of each operation by copying the initial value of each global variable to its shadow copy. Second, it enables the exception handling of the supervisor call (`SVC`), memory management fault, and bus fault. Finally, it drops the privilege and runs the unprivileged application code to ensure privilege isolation.

5.2 Resource Isolation

Figure 6 shows the memory layout and the access permissions of the final image generated by `OPEC-Compiler`. The data of `OPEC-Monitor` and the relocation tables of each operation are placed in the memory region that is only writable at the privileged level. It prevents the isolation policy from being tampered with by a compromised unprivileged operation. The operation metadata excluding relocation tables is stored in the read-only memory.

The eight MPU regions are arranged as follows. Region 0 sets all memory ranges as read-only. Region 1 enables the execution of the unprivileged application code. Region 2 and 3 enforce read and write permissions to the stack and the operation data section. Region 4 to 7 are reserved for peripheral access. In the following, we will present how our system enforces resource isolation at runtime.

Global Variables `OPEC-Monitor` synchronizes shared variables in the *operation data section* and modifies the variables relocation table to ensure that the program accesses right shadow copies. The data synchronization process during the operation switch is illustrated in Figure 7. When entering into or returning from one operation, `OPEC-Monitor` first identifies whether the global variable needs to be synchronized. As shown in Figure 7(b)(c), global variable d is an external variable to operation B and C. Therefore, d and e should be synchronized when entering into operation C

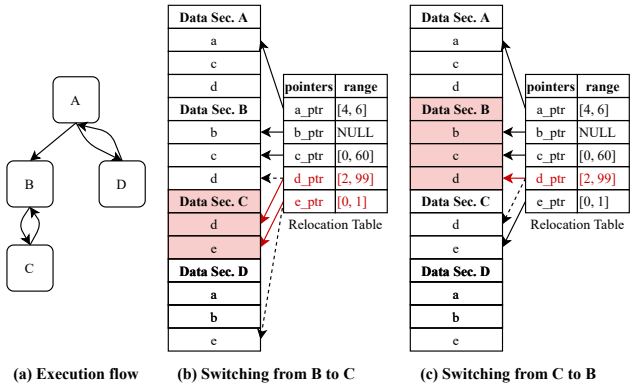

Figure 7. External global variables synchronization.

from B but only *d* needs to be synchronized when returning back to operation B. Before synchronizing, `OPEC-Monitor` performs data sanitization to check whether the value is legitimate by comparing the value with the developer-provided valid value range. If the check fails, there may exist data corruption inside the operation. The `OPEC-Monitor` will abort the program and prevent the value of the corrupted shadow copy from propagating to other operations. This sanitization ensures that new values are in safe ranges specified by developers. For instance, the speed of a moving robotic arm should be in a limited range.

Stack The challenge of stack protection stems from accessing local variables on the previous stack frame of another operation. ACES [18] handles this challenge by using a micro-emulator. It uses one MPU region to enforce the access permission for the stack and disables as many sub-regions of this MPU region as possible to prevent access to previous portions of the stack. When the program accesses the previous portion of the stack, a memory access fault occurs. Then it invokes the micro-emulator to check against an allow list and enables this access. However, this solution requires to profile the program to get the precise stack access information, which may be imprecise.

OPEC implements precise protection for the stack by leveraging the sub-region feature of the MPU and the data relocating technique. In our design, the stack is protected by an individual MPU region. `OPEC-Monitor` divides the stack into eight equally sized portions. Each portion of the stack corresponds to a sub-region of the MPU region. `OPEC-Compiler` analyzes the parameters of the entry function of each operation to get the size of the arguments saved on the stack and the size of data pointed by the pointer-type arguments. If these data need to be accessed by the new operation but saved in the previous operation's stack, `OPEC-Monitor` will relocate them when switching to a new operation.

Specifically, `OPEC-Monitor` first copies the data pointed by the pointer-type arguments along with the arguments saved on the stack to the first available sub-region of the stack. Next, it redirects the pointer-type arguments to point to their duplicated copies on its stack. `OPEC-Monitor` disables the previous sub-regions of the stack to protect the stack from being corrupted by other operations. Then it saves the previous stack pointer and updates its value. Finally, `OPEC-Monitor` disables the sub-regions corresponding to the previous stack frame to prevent access.

Figure 8 shows an example of the data relocation process during the operation switching. The default operation has a local variable `buf` which is saved at address `0x2F88`. Before entering into the operation `Foo`, the last two arguments of the entry function are pushed into the stack (others are saved in registers). One is the start address of `buf`, another is its size. Note that an operation can use multiple sub-regions of the stack. This strategy significantly mitigates the issue of insufficient stack space. In Figure 8(c), the operation `Foo` occupied three sub-regions. After exiting from the `Foo`, `OPEC-Monitor` copies back the data pointed by pointer-type arguments and restores the stack pointer as shown in Figure 8(e).

Peripherals OPEC reserves four MPU regions for each operation to access four peripherals and protect every peripheral by an individual MPU region. However, one operation may need more than four MPU regions for accessing peripherals because: 1) the operation may need to access multiple peripherals; 2) one peripheral may need two more MPU regions due to the MPU region alignment requirement. To solve this problem, `OPEC-Monitor` virtualizes the MPU regions. `OPEC-Monitor` uses the memory management fault handler to update the MPU configurations dynamically. Before updating the MPU, `OPEC-Monitor` verifies whether it is legitimate access by checking the peripheral address against the peripheral list of the current operation. If so, `OPEC-Monitor` will select one of the four reserved MPU regions and update its configurations. Note that `OPEC-Monitor` uses the round-robin algorithm to determine which MPU region should be swapped out.

For core peripherals that trigger a bus fault when accessed by unprivileged code, `OPEC-Monitor` emulates the execution of the `store`/`load` instructions. Similarly, `OPEC-Monitor` utilizes the bus fault handler to handle the bus fault triggered by those instructions. `OPEC-Monitor` first checks whether the fault is triggered by unprivileged access to core peripherals. Then it examines whether the address (the target or source address of the `store` or `load` instructions) causing this fault is permitted to access by the current operation. Finally, `OPEC-Monitor` emulates the `load` or `store` instructions to read from or write to values to those core peripherals.

Heap `OPEC-Compiler` cannot determine which part of the heap memory is used by a function statically. Therefore, if one function uses a variable that resides in the heap memory, the whole heap memory is allowed to be accessed by this function. Moreover, `OPEC-Monitor` places the heap memory into a separate section rather than operation data sections,

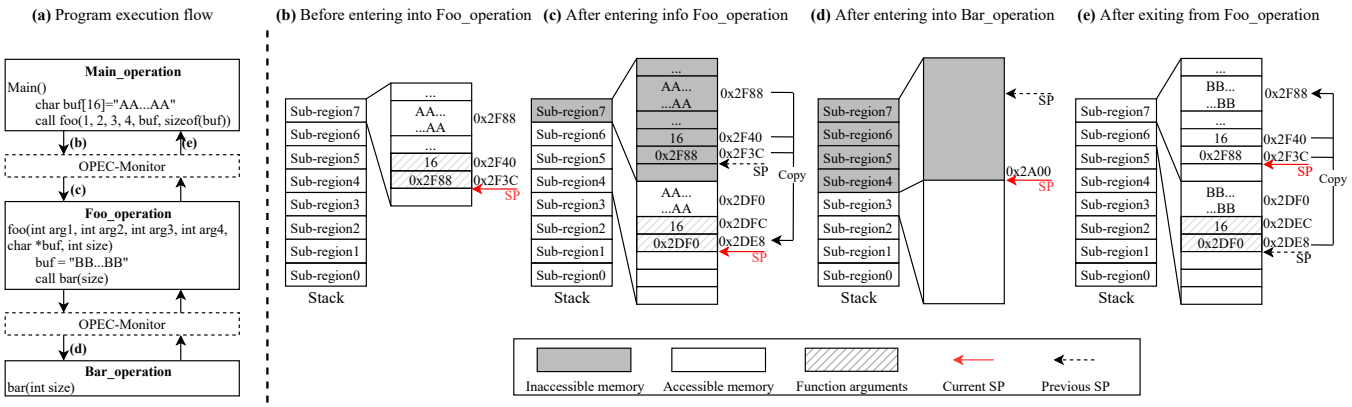

Figure 8. Illustration of stack protection by OPEC.

which reduces performance overhead by avoiding copying or writing back its content when switching the operation.

5.3 Operation Switch

Operation switch is triggered by executing `SVC` instructions before and after the call site of each operation entry function. Then `OPEC-Monitor` handles the `SVC` and executes the operation switch routine. `OPEC-Monitor` uses the *operation stack* to save the context of previous operations so that the context can be restored. The operation context information includes MPU configurations, the stack pointer value, and the peripheral list. This operation stack is only writable at the privileged level, protecting it from unauthorized changes.

When entering into a new operation, `OPEC-Monitor` copies global shadow variables and relocates variables on the stack. In the meanwhile, `OPEC-Monitor` examines the pointers field information recorded by `OPEC-Compiler` (4.2). Specifically, `OPEC-Monitor` checks whether these pointers point to the shadow variables of another operation's data section. If so, `OPEC-Monitor` redirects those pointers to the shadow variables of the new operation. After that, it configures the MPU and returns to the first instruction of the new operation.

When exiting from one operation, `OPEC-Monitor` sanitizes and synchronizes the shadow variables of the current operation. Next, it recovers the stack for the previous operation and clears the value of general-purpose registers. Finally, it gets the saved context information of the target operation from the *operation stack* and restores the MPU configuration.

6 Evaluation

In this section, we evaluate our OPEC prototype to answer the following three research questions:

- R1: What are the security benefits provided by OPEC?
- R2: What is the performance overhead of OPEC?
- R3: What are the advantages of OPEC compared to the state-of-the-art security isolation?

We generate two different binaries for each tested application. The first one serves as the baseline that is built without any protection, and the second one is built with OPEC. To evaluate our system, we first use the PinLock application as the case study to illustrate how OPEC constrain a compromised operation, whereas ACES cannot. Then, we evaluate the effectiveness of the security isolation of OPEC through several static metrics. Next, we measure the performance overhead induced by OPEC and compare it with ACES to show how lightweight OPEC is. After that, we evaluate the partition-time and execution-time over-privilege issues of OPEC and ACES. Finally, we measure the efficiency of resolving the icalls.

We test OPEC with six representative IoT applications and the CoreMark benchmark [4] on two boards, a STM32F4-Discovery [10] development board and a STM32479I-EVAL evaluation board [8]. Both boards have an ARM Cortex-M4 core. The STM32479I-EVAL board has diverse peripherals, such as a camera and an ethernet interface. Whereas five applications are implemented by STMicroelectronics [9], PinLock is adopted from the tested applications used by ACES [18]. The description of these tested applications is available at this site [1].

6.1 PinLock Case Study

We use the PinLock application as an example to demonstrate how OPEC solves the partition-time over-privilege issue and protects the system from a compromised compartment. As illustrated in Listing 1, the PinLock has six tasks. Virtually, task `Unlock_Task` and `Lock_Task` are two independent tasks that should be divided two into different compartments (or operations). Both `Lock_Task` and `Unlock_Task` invoke the function `HAL_UART_Receive_IT`, which is defined in the vendor-provided Hardware Abstraction Libraries (HAL), to receive a user input pin from the serial port. *We assume there is a vulnerability in the function* `HAL_UART_Receive_IT`. An attacker with the arbitrary memory write ability can exploit this vulnerability to overwrite the correct pin. The pin is finally saved to a global buffer `PinRxBuffer`. For the `Unlock_Task` function, it hashes the global variable `PinRxBuffer`. The hash result is then compared to a global variable `KEY`, which is the hash value of the correct pin. If the user input pin is correct,

Table 1. The metrics of the security evaluation. The percent in the parentheses is compared to the baseline; a smaller number is better. **#OPs.**: the number of operations. **#Avg. Funcs**: the average number of functions in an operation. **#Pri. Code**: the size of code runs in the privileged level. **#Avg. GVars**: the average size of the accessible global variables.

Application	#OPs	#Avg. Funcs	#Pri. Code(%)	#Avg. GVars(%)
PinLock	6	14.00	8344(8.86)	72.86(44.43)
Animation	8	85.00	8398(5.10)	1422.44(35.61)
FatFs-uSD	10	77.00	8364(7.39)	1134.09(52.07)
LCD-uSD	11	67.82	8362(4.55)	1126.50(34.19)
TCP-Echo	9	57.67	8646(5.76)	14989.2(45.82)
Camera	9	61.11	8428(3.75)	1428.40(28.52)
CoreMark	9	17.11	8414(12.87)	1100.50(48.10)
Average	8.86	54.24	8422.29(6.90)	3039.14(41.25)

the function `do_unlock` will be called. For the `Lock_Task` function, it checks whether the first byte of the `PinRxBuffer` is 0 and performs locking by invoking the function `do_lock`. Therefore, the global variable `PinRxBuffer` is shared by these two compartments.

To save the MPU regions usage, ACES groups the two global variables `PinRxBuffer` and `KEY` into one data region. It results in the partition-time over-privilege issue, i.e., task `Lock_Task` could access the unnecessary global variable `KEY`. If `Lock_Task` is compromised, attackers are capable of overwriting the variable `KEY` and perform unlocking directly to break the isolation. However, OPEC avoids this issue. The operation data section of `Lock_Task` does not contain the shadow copy of `KEY`. When the application executes operation `Lock_Task`, attackers cannot overwrite the variable `KEY`, which ensures security isolation.

6.2 Security Evaluation

Privileged Code OPEC enforces privilege isolation by running the application code at the unprivileged level, whereas running `OPEC-Monitor` at the privileged level. Note that all the code executes at the privileged level in baseline because there is no privilege isolation. OPEC reduces the privileged code running at the privileged level. As the results show in Table 1, the average percentage of the privileged code size is 6.60% compared to the baseline.

As we discussed in Section 5.2, reading or writing the core registers requires the code to run at the privileged level. To solve this issue, ACES lifts the compartment to the privileged level if this compartment needs to access the core peripherals, which executes application code at the privileged level.

Accessible Global Variables We measure the accessible global variable size of each operation to understand the effectiveness of resource isolation. As shown in Table 1, OPEC can effectively confine the average percentage of accessible global variables of each operation to 41.25%. Specifically, FatFs-uSD's average percentage of accessible global variables is large. FatFs-uSD implements a FatFs file system on a SD card to manage its storage. The program has two large structure variables, `MyFile` and `SDFatFs` used as a file object and a file system object, respectively. These two variables are shared among several operations, which leads to the large average percentage of this metric of this application. Despite this case, our system effectively confines the operations' accessible global variables of other applications.

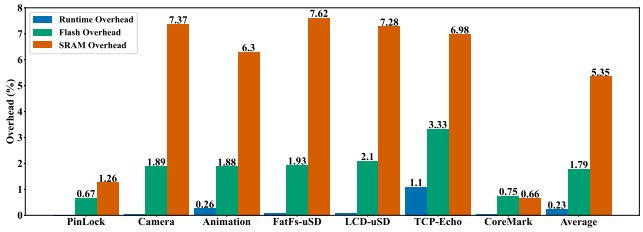

Figure 9. Performance overhead of OPEC.

6.3 Performance Overhead

Runtime Overhead Since bare-metal embedded systems are resource-constrained devices, a lightweight security isolation scheme can protect the availability of the devices and save energy. To measure the runtime overhead caused by OPEC, we record the timestamp before and after executing the code. The executed time is the difference value between these two timestamps. We take advantage of a peripheral named Data Watchpoint and Trace (DWT) [3] to measure the runtime overhead. In particular, DWT measures the number of used clock cycles at a specific point. We first record the number of clock cycles used by the baseline and the application built with OPEC. After that, the latter is further divided by the former to calculate the final ratio. Note that DWT does not influence the execution time of applications. As illustrated in Figure 9, the average runtime overhead induced by OPEC is 0.23% and the maximum overhead is 1.1%. The results show that OPEC has a negligible runtime overhead since OPEC adopts a operation-based partition methodology that follows the execution flow of the application, which avoids frequent domain switching.

Flash Overhead To measure the overall increased memory, we use a tool readelf [7] to collect each section's memory information of the final binary. The result is further divided by the Flash size of the device to get the increased memory ratio. The Flash size of the STM32F4-Discovery board and the STM32479I-EVAL board is 1 MB and 2 MB, respectively. As shown in Figure 9, OPEC causes a minimal Flash overhead. The average Flash overhead induced by OPEC is 1.79%, and the maximum is 3.33%. Note that the operation metadata used by the `OPEC-Monitor` accounts for the most Flash overhead to six applications.

SRAM Overhead We measure the SRAM overhead by calculating the increased memory caused by operation data sections. The final results are divided by the size of the SRAM.

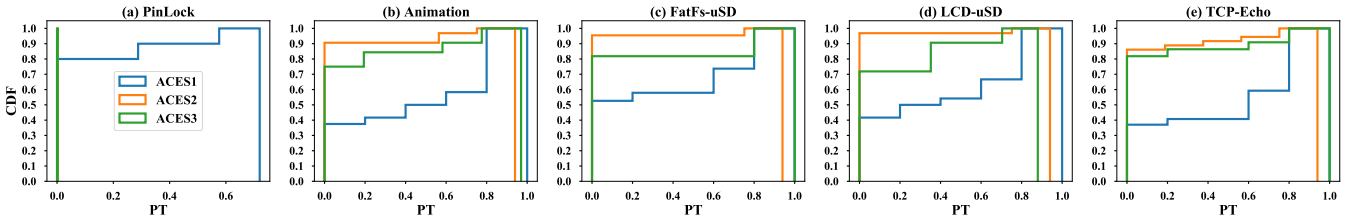

Figure 10. The results of partition-time over-privilege value (PT) of the five applications tested in ACES under three partitioning strategies. The legends in the last four figures are the same as those in the first one. **CDF**: Cumulative Distribution Function.

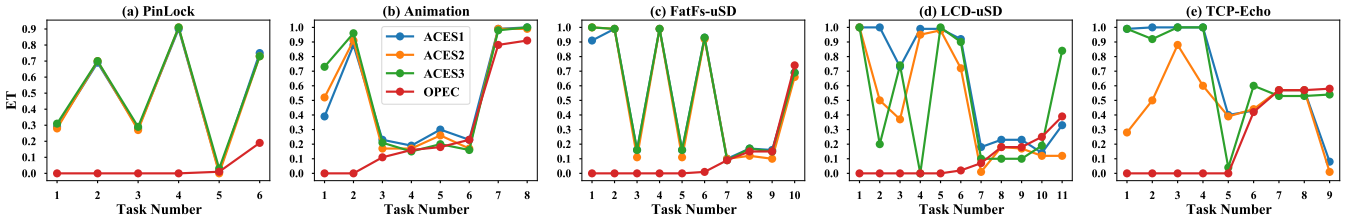

Figure 11. The evaluation results of execution-time over-privilege value (ET) of the five applications tested in ACES under three partitioning strategies. The legend in the first and last three figures are the same as those in the second one.

The SRAM size of the STM32F4-Discovery board and the STM32479I-EVAL board is 192 KB and 288 KB, respectively. As illustrated in Figure 9, OPEC causes a moderate SRAM overhead. The maximum SRAM overhead is 7.62%. The average SRAM overhead is 5.35%. The operation data sections and their fragments required by the MPU region account for the most SRAM overhead. As discussed in Section 2, the MPU region size must be the power of two. This limitation leads to the internal fragments of operation data sections, i.e., one operation data section probably contains some unused memory. Such SRAM overhead is hard to eliminate.

6.4 Comparison to ACES

We compare OPEC with ACES [18], the state-of-the-art security isolation, for the over-privilege issues and performance overhead. We use five applications evaluated by ACES for comparison and evaluate the three supported partition strategies, i.e., filename (ACES1), filename without optimization (ACES2), and peripheral (ACES3).

Partition-time Over-privilege is caused by shared global variables between compartments and the limitation of the MPU. We use a new metric, partition-time over-privilege value (PT for short), to quantitatively measure the degree of the partition-time over-privilege issue. The PT of a domain (a compartment or an operation) is the percentage of the size of the global variables that are *unneeded* by it to the size of its accessible global variables. We consider one global variable *unneeded* by a domain if and only if there is no function within the domain that has a data dependency on this variable. The PT value can be calculated by this equation:

$$PT = \frac{\sum_i^n var2size(unneeded\_var_{fi})}{\sum_i^n var2size(accessible\_var_{fi})} \quad (1)$$

where f_i is a function of a domain, n is the number of functions that a domain contains, and $var2size$ is the function that calculates the size of a set of variables. If a domain does not access any global variable or it does not suffer from the partition-time over-privilege issue, its PT is 0.

We evaluate the PT of each compartment of the five applications used by ACES [18] under the three different partition strategies. Figure 10 presents the cumulative distribution of PT for the compartments of different applications. The results reveal that all strategies supported by ACES, except for the application PinLock partitioned by the strategy ACES2 and ACES3, are suffering from the partition-time over-privilege issue. Note that our system does not have this issue because OPEC adopts the global variables shadowing technique (4.4), which avoids incorporating unneeded global variables into the operation data section.

Execution-time Over-privilege is caused by including unnecessary code when executing a task. Similarly, we use a new metric, execution-time over-privilege value (ET for short), to measure the degree of the execution-time over-privilege of a task. The basic idea is to calculate the used global variables when running a specific task. The ET of a task is the percentage of the size of the global variables that are actually *unused* by it during execution to the size of its *needed* global variables. If one function is executed in that task, the global variables needed by this function are *used*. Otherwise, they are *unused*.

To evaluate the execution-time over-privilege of a task under different partitioning strategies, we need to measure the size of its used global variables and its needed global variables. A task's used global variables are all the global data dependency of the functions executed in the task. For OPEC, since each operation is actually a task, the needed global variables of a task are all the global data dependency of the operation. For ACES, the needed global variables are all the global data dependency of the functions within the compartments involved during execution. We can calculate

Table 2. Comparison of **R**untime **O**verhead, **F**lash **O**verhead, and **S**ram **O**verhead, between OPEC and ACES.

Application	Policy	RO(X)	FO(%)	SO(%)
PinLock	**OPEC**	1.00	0.74	1.40
	ACES-1	1.00	1.03	0.28
	ACES-2	1.01	0.94	0.25
	ACES-3	1.00	0.99	0.18
Animation	**OPEC**	1.00	1.88	6.30
	ACES-1	1.19	2.51	1.21
	ACES-2	5.70	3.53	4.30
	ACES-3	1.13	2.71	0.95
FatFs-uSD	**OPEC**	1.00	1.92	7.62
	ACES-1	1.90	1.57	0.43
	ACES-2	1.95	2.04	0.43
	ACES-3	1.25	1.59	0.42
LCD-uSD	**OPEC**	1.00	2.10	7.28
	ACES-1	1.78	2.52	4.44
	ACES-2	5.69	2.90	0.91
	ACES-3	1.72	2.10	1.24
TCP-Echo	**OPEC**	1.01	3.65	6.56
	ACES-1	2.17	3.67	0.64
	ACES-2	3.69	0.94	6.28
	ACES-3	1.22	0.99	0.49

the ET of a task under a specific partitioning methodology by the following equation:

$$ET = 1 - \frac{\sum_i^n var2size(used\_var_{di})}{\sum_i^n var2size(needed\_var_{di})} \quad (2)$$

where d_i is the domain that a task involves, n is the number of domains that a task involves.

To acquire the executed functions within a task, we need to trace the execution of the tested applications. Note that we need to trace the execution at the function granularity. We use the GDB [5] to single-stepping the whole program. Specifically, we use a script for GDB to automatically execute the tested application step by step and record the execution information, such as the line number of the source code to the logs. After that, we parse the logs and extract the executed functions.

As the results shown in Figure 11, our system effectively mitigates the execution-time over-privilege issue. However, there are some tasks in **LCD-uSD** and **TCP-Echo** partitioned by OPEC that have higher ET value than ACES. The reason is that those tasks contain more unnecessary code, which can be divided into two categories. The first category is *untaken branch*. This category is prevalent, including error handling code and unmatched peripheral configuring functions. The second category is *spurious icall target*. Due to the over-approximation of the point-to analysis, there are false positives when resolving the icalls. Thus these functions will not execute at runtime.

Table 3. Efficiency of icall analysis. **#Icall**: the number of icalls. **#SVF**: the number of resolved icalls by the point-to analysis. **Time(s)**: the time used to perform the point-to analysis. **#Type**: the number of resolved icalls by the type-based analysis. **#Avg.**: the average number of targets of the icalls. **#Max**: the maximum number of targets of the icalls.

Application	#Icall	#SVF	Time(s)	#Type	#Avg.	#Max
PinLock	1	0	0.06	1	1.00	1
Animation	18	12	0.34	6	1.33	2
FatFs-uSD	18	12	0.25	6	1.72	3
LCD-uSD	19	12	0.33	0	0.58	1
TCP-Echo	29	15	5.26	13	1.28	5
Camera	35	30	6.45	5	1.77	5
CoreMark	1	1	0.12	0	2.00	2

Performance Overhead As illustrated in Table 2, OPEC incurs overall lower runtime overhead than ACES. It is precisely because of the operation-based partition methodology that follows the program execution flow and avoids frequent domain switching. Moreover, the Flash overhead of OPEC is slightly smaller than the ACES. Our system incurs more SRAM overhead than ACES because OPEC uses a global variable shadowing technique. On the contrary, ACES moves global variables and merges them into a separate region, which does not increase global variables.

6.5 Efficiency of the Icall Analysis

This section evaluates the efficiency of our icall analysis because it affects building a precise function call graph which is essentially important in the operation partitioning stage (Section 4). Table 3 presents the results. In summary, the time cost for performing the point-to analysis via SVF is less because all the tested applications have a small code size. As previously discussed, for those icalls that cannot be resolved by SVF, we use the type-based analysis to analyze the potential targets. Among the applications, the maximum number of targets of those icalls is 5 (in application **TCP-Echo** and **Camera**). For the average targets of the icalls, the maximum average is 1.77 (in application **Camera**). The results show that the false positives of the icalls' targets have a limited impact.

7 Discussion and Future Work

Confused Deputy Attacks Attackers may compromise and manipulate an ordinary operation to feed malicious inputs to a critical operation, such as the `Unlock_Task` operation of PinLock, to trigger the execution of security-sensitive functions. Security isolation is vulnerable to this kind of attacks [16] [22]. OPEC provides data sanitization to shared variables between operations to mitigate such attacks.

Defects of Static Analysis Our system uses static analysis to build a sound function call graph and to determine the global data and peripherals dependency of each function. For icalls and implicit data usage through pointers, OPEC leverages the point-to analysis to find out the potential legitimate targets of icalls and data pointers. In particular, we use the SVF [34] to conduct the point-to analysis. However, the SVF guarantees soundness but sacrifices completeness, i.e., the results are over-approximated and contain false positives. Although it may introduce the execution-time over-privilege issue, it avoids program execution errors caused by missing dependency. The static analysis is not the contribution of our work and the improvements of the static analysis could be transparently integrated into OPEC.

Heap As discussed in Section 5.2, OPEC supports dynamic memory allocation, i.e., heap. However, if the program uses the heap implemented by the static libraries, e.g., glibc [6], `OPEC-Compiler` cannot identify such access. If the program uses such kind of heap, it needs to be modified by replacing dynamically allocated memory objects with global variables. In our experiment, four-sevenths of tested applications use the heap implemented by the static libraries to allocate memory. Replacing heap objects with global variables takes trivial efforts. In the future, we will extend our resource dependency analysis to these statically linked libraries.

Multithreading Our system is aimed to provide security isolation to bare-metal embedded systems, which execute single monolithic programs with a single thread. As a result, OPEC cannot be directly applied to embedded systems with multithreading. In the future, we will improve the shadowing data technique to support multithreading.

8 Related Work

Besides being related to the research on enforcing security isolation on bare-metal embedded systems, OPEC is also associated with the work towards providing security defenses and ensuring the correct execution of the embedded systems.

Security Isolation Clements et al. [19] proposes EPOXY, which enforces privilege isolation on bare-metal embedded systems by executing sensitive instructions at the privileged level. However, EPOXY neglects resource isolation. Kim et al. [25] proposes MINION, which extends the previous work and enforces thread-level memory isolation on real-time microcontroller systems. Furthermore, Clements et al. [18] proposes ACES that enforces privilege isolation and sub-thread level memory isolation. Huo et al. [24] extends ACES by adopting a machine learning-assisted compartmentalization scheme. TrustLite [27] introduces execution-aware MPU to isolate and prevent trusted components from unauthorized modifications. TyTAN [15] provides security isolation by leveraging the hardware extensions proposed by TrustLite. However, they require a customized hardware extension, which is not available on real hardware. There are other works enforces security isolation on commodity software [17, 21, 29, 30, 36, 37]. OPEC proposes privilege isolation and improved resource isolation to bare-metal embedded systems.

Security Defenses A large number of systems intended to deploy protection or mitigation techniques widely used on desktop systems to deeply embedded systems. EPOXY [19] uses DEP and a modified SafeStack to prevent control-flow hijacking attacks. It also uses diversification to introduce uncertainty to the produced firmware, which raises the cost of attackers to analyze the firmware. Abbasi et al. [12] quantitatively investigate the mitigation techniques adopted in deeply embedded systems. It further proposes uArmor that implements DEP and stack canaries for those systems. Kwon et al. [28] realizes an efficient eXecute-Only-Memory on Cortex-M microcontrollers by leveraging the unprivileged memory instructions of the ARMv7-M architecture. Zhou et al. [42] also takes advantage of this feature to design a shadow stack on low-end embedded systems. Almakhdhub et al. [14] proposes uRAI to protect the return address integrity of microcontroller-based embedded systems. Moreover, it enforces software fault isolation (SFI) on the privileged exception handlers. These security defenses are orthogonal to the security isolation provided by OPEC. They can be integrated in to our system to provide protection to the code inside one operation.

Correctness Insurance There is some research focuses on diagnosing embedded systems. Kim et al. [26] proposes MAYDAY to facilitate the post-accident analysis of drones. Fang et al. [20] uses the record and replay technique to troubleshoot the errors of embedded systems. Niesler et al. [32] exploits a hardware feature named Flash Patch and Breakpoint unit to patch the embedded real-time systems at runtime. Similar works [38–40] also introduce hot patching to embedded systems. Xu et al. [41] and Huber et al. [23] aim to recover the embedded systems even when they are compromised. Another category of research enables remote attestation to ensure the code integrity, control-flow integrity, and data integrity of the embedded systems [13, 33, 35]. OPEC enforces the operation isolation on bare-metal embedded systems and constrains the compromised component.

9 Conclusion

In this paper, we propose operation-based security isolation for bare-metal embedded systems. We prototype our system, OPEC, which enforces privilege isolation and resource isolation. OPEC overcomes the partition-time over-privilege issue and mitigates the execution-time over-privilege issue of existing security isolation schemes on bare-metal embedded systems through global data shadowing and operation-based partitioning. The evaluation demonstrates the security benefits. OPEC induces low runtime overhead and moderate memory overhead.

References

[1] 2021. AnonymousSubmission185. https://github.com/AnonymousPaperCodeRepo/AnonymousSubmission185.

[2] 2021. ARMv7-M Architecture Reference Manual. https://developer.arm.com/documentation/ddi0403/latest.

[3] 2021. Data Watchpoint and Trace Unit. https://developer.arm.com/documentation/ddi0439/b/Data-Watchpoint-and-Trace-Unit.

[4] 2021. EEMBC, "Coremark - industry-standard benchmarks for embedded. https://www.eembc.org/coremark/.

[5] 2021. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/.

[6] 2021. GNU Arm Embedded Toolchain. https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm.

[7] 2021. readelf. https://man7.org/linux/man-pages/man1/readelf.1.html.

[8] 2021. STM32479I-EVAL Evaluation Board. https://www.st.com/en/evaluation-tools/stm32479i-eval.html.

[9] 2021. STM32Cube MCU Full Package. https://github.com/STMicroelectronics/STM32CubeF4/tree/master/Projects/STM32469I_EVAL.

[10] 2021. STM32F4-Discovery Board. https://www.st.com/en/evaluation-tools/stm32f4discovery.html.

[11] 2021. The LLVM Compiler Infrastructure. https://llvm.org/.

[12] Ali Abbasi, Jos Wetzels, Thorsten Holz, and Sandro Etalle. 2019. Challenges in designing exploit mitigations for deeply embedded systems. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. 31–46.

[13] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. C-FLAT: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 743–754.

[14] Naif Saleh Almakhdhub, Abraham A Clements, Saurabh Bagchi, and Mathias Payer. 2020. uRAI: Securing Embedded Systems with Return Address Integrity. In *Network and Distributed Systems Security (NDSS) Symposium*.

[15] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. 2015. TyTAN: Tiny trust anchor for tiny devices. In *Proceedings of the 52nd annual design automation conference*. 1–6.

[16] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. 2012. Towards Taming Privilege-Escalation Attacks on Android. In *NDSS*, Vol. 17. 19.

[17] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-Grained Execution Units with Private Memory. In *2016 IEEE Symposium on Security and Privacy*. 56–71.

[18] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. 2018. ACES: Automatic Compartments for Embedded Systems. In *Proceedings of the 27th USENIX Security Symposium*. 65–82.

[19] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. 2017. Protecting bare-metal embedded systems with privilege overlays. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*. 289–303.

[20] Kaiming Fang and Guanhua Yan. 2020. IoTReplay: Troubleshooting COTS IoT Devices with Record and Replay. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. 193–205.

[21] Khilan Gudka, Robert NM Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G Neumann, and Alex Richardson. 2015. Clean application compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1016–1031.

[22] Norm Hardy. 1988. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review* 22, 4 (1988), 36–38.

[23] Manuel Huber, Stefan Hristozov, Simon Ott, Vasil Sarafov, and Marcus Peinado. 2020. The Lazarus Effect: Healing Compromised Devices in the Internet of Small Things. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 6–19.

[24] Dongdong Huo, Chao Liu, Xiao Wang, Mingxuan Li, Yu Wang, Yazhe Wang, Peng Liu, and Zhen Xu. 2020. A Machine Learning-Assisted Compartmentalization Scheme for Bare-Metal Systems. In *International Conference on Information and Communications Security*. 20–35.

[25] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. 2018. Securing Real-Time Microcontroller Systems through Customized Memory View Switching. In *Network and Distributed Systems Security (NDSS) Symposium*.

[26] Taegyu Kim, Chung Hwan Kim, Altay Ozen, Fan Fei, Zhan Tu, Xiangyu Zhang, Xinyan Deng, Dave Jing Tian, and Dongyan Xu. 2020. From Control Model to Program: Investigating Robotic Aerial Vehicle Accidents with MAYDAY. In *29th USENIX Security Symposium (USENIX Security 20)*. 913–930.

[27] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. 2014. TrustLite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–14.

[28] Donghyun Kwon, Jangseop Shin, Giyeol Kim, Byoungyoung Lee, Yeongpil Cho, and Yunheung Paek. 2019. uXOM: Efficient eXecute-Only Memory on ARM Cortex-M. In *28th USENIX Security Symposium (USENIX Security 19)*. 231–247.

[29] Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. 2019. Program-mandering: Quantitative privilege separation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1023–1040.

[30] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1607–1619.

[31] Kangjie Lu and Hong Hu. 2019. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1867–1881.

[32] Christian Niesler, Sebastian Surminski, and Lucas Davi. 2021. HERA: Hotpatching of Embedded Real-time Applications. In *Proceedings of THE 28th Network and Distributed System Security Symposium*.

[33] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. 2020. APEX: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise. In *29th USENIX Security Symposium USENIX Security 20)*. 771–788.

[34] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266.

[35] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. 2020. OAT: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1433–1449.

[36] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. 2017. Towards fine-grained, automated application compartmentalization. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*. 43–50.

[37] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *NDSS*.

[38] Michael Wahler and Manuel Oriol. 2014. Disruption-free software updates in automation systems. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*. 1–8.

[39] Michael Wahler, Stefan Richter, Sumit Kumar, and Manuel Oriol. 2011. Non-disruptive large-scale component updates for real-time controllers. In *2011 IEEE 27th International Conference on Data Engineering Workshops*. 174–178.

[40] Michael Wahler, Stefan Richter, and Manuel Oriol. 2009. Dynamic software updates for real-time systems. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*. 1–6.

[41] Meng Xu, Manuel Huber, Zhichuang Sun, Paul England, Marcus Peinado, Sangho Lee, Andrey Marochko, Dennis Mattoon, Rob Spiger, and Stefan Thom. 2019. Dominance as a new trusted computing primitive for the internet of things. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1415–1430.

[42] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J Walls. 2020. Silhouette: Efficient protected shadow stacks for embedded systems. In *29th USENIX Security Symposium (USENIX Security 20)*. 1219–1236.