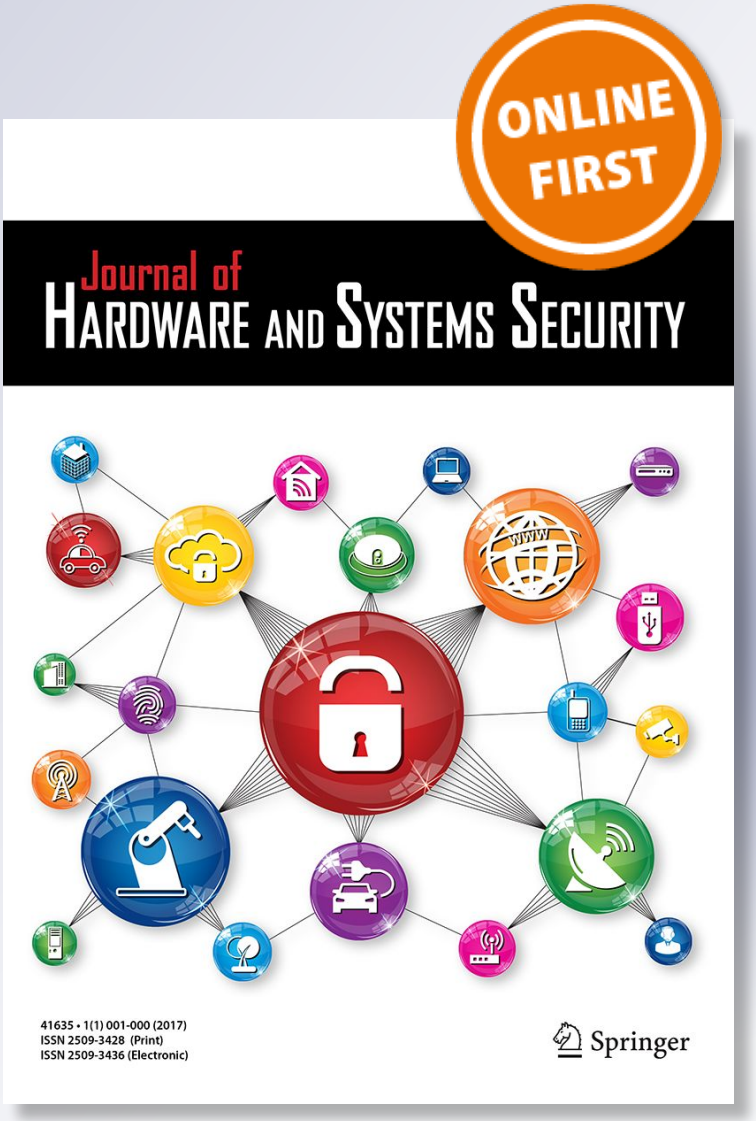
An In-depth Study of MPU-Based Isolation Techniques

Abderrahmane Sensaoui, Oum-El-Kheir Aktouf, David Hely & Stephane Di Vito

Journal of Hardware and Systems Security

ISSN 2509-3428

J Hardw Syst Secur
DOI 10.1007/s41635-019-00078-6



Springer

Journal of Hardware and Systems Security
https://doi.org/10.1007/s41635-019-00078-6

An In-depth Study of MPU-Based Isolation Techniques

Abderrahmane Sensaoui[1,2] · Oum-El-Kheir Aktouf[2] · David Hely[2] · Stephane Di Vito[1]

Received: 26 July 2018 / Accepted: 23 September 2019

Abstract
Many attacks have been reported and published targeting constrained embedded systems. Attackers try to exploit vulnerabilities through all possible layers of abstraction. A single vulnerability can be enough to take over the whole device and change its intended behavior. Hardware/software isolation architectures implemented in embedded devices provide access control mechanisms to establish a protected execution environment and guarantee the behavior of the running applications. They enforce the boundaries to stop a malicious flaw from propagating from one application to others, especially those that are critical. They represent a form of resilience to different exploits. This paper provides a detailed study of existing memory protection unit-based isolation architectures for lightweight devices and defines four important criteria to evaluate and compare architectures from both academia and industry. Outcomes of this work will help developers and hardware designers to find balance between performance and security.

Keywords Memory protection · Isolation · Security software and hardware · Performance evaluation · Trusted computing · Survey

1 Introduction

The Nintendo Switch was attacked several times by different security research teams since its release. Two exploits have been published, *Fusee Gelee* [1] and *ShofEL2* [2], which both take advantage of a buffer overflow vulnerability found in the USB stack. This allows execution of arbitrary code and can lead, for example, to the system running unofficial games and to the attacker customizing the firmware.

Computer security has always been prone to software attacks. Countermeasures are developed by researchers and programmers to mitigate threats; however, attackers constantly discover a way to bypass these defenses and compromise devices. Some protection solutions, like, for example, control-flow integrity and data-flow integrity [3], have been proven effective, but they are not deployable by the industry because they may require CPU changes [4–6]. Thus, industrials, first, have to change the existing hardware and, second, may need other licenses to customize the hardware which can be very costly.

Researchers and developers, from both industry and academia, have placed a lot of effort into security research in the previous years to develop resilient architectures which comply with industrial constraints. Isolation has been around for decades now via Multics (Multiplexed Information and Computing Service) [7]. Multics provides computer multitasking, where each process has its own memory space, resulting in isolated environments. Therefore, a flaw in one environment will not affect other environments. Since then, lots of security isolation architectures have been developed. These architectures are based on privilege separation. So, some components run in a high-privilege mode and are usually responsible for establishing isolation within the system. They are also responsible for controlling the conditions under which an application can have access to specific resources. Other components run in a low-privilege mode, they are untrusted, and they have limited access to resources.

To develop such an environment, many techniques have been proposed, but memory protection is one of

✉ Abderrahmane Sensaoui
abderrahmane.sensaoui@lcis.grenoble-inp.fr

Oum-El-Kheir Aktouf
oum-el-kheir.aktouf@lcis.grenoble-inp.fr

David Hely
david.hely@lcis.grenoble-inp.fr

Stephane Di Vito
stephane.divito@maximintegrated.com

[1] Maxim Integrated, La Ciotat, France

[2] University of Grenoble Alpes, Grenoble INP, LCIS, Grenoble, France

Published online: 22 November 2019

the most common and used techniques in computer security. Currently, many architectures are based on the memory management unit (MMU). The MMU is a hardware component that provides virtual memory and memory protection at the same time. However, the MMU is not suitable for all processors, especially for small and low-power devices. Lightweight devices are extremely resource-constrained, and they do not need complex memory management, so they do not embed an MMU, especially if the deterministic and time-critical aspects cannot be guaranteed. Many mechanisms have been proposed for lightweight devices [8–14], and the memory protection unit (MPU) is one of them. It is a hardware component that protects memory and controls accesses. It was designed for devices that do not require complex memory management.

The objective of this paper is to study, evaluate, and compare some MPU-based isolation architectures for lightweight devices. To better understand how MPU-based isolation could increase the resilience of embedded devices, we provide an analysis of four different dimensions (e.g., security features, security protection, performance, and memory consumption) of four different architectures. These architectures were selected from industry and academia. The four chosen ones are the widest spread and have enough available data for us to conduct our study.

The contributions of the paper are:

- Different evaluation criteria are defined to evaluate the studied architectures;
- Many weaknesses and vulnerabilities from various papers and from the National Vulnerability Database [15] are analyzed to extract the most common attack vectors to evaluate their impact on isolation-based architectures;
- Different isolation architectures are studied, and our choice was to select five MPU-based architectures among many;
- Finally, these architectures are evaluated and discussed based on the proposed evaluation criteria.

This paper is structured as follows. The "Evaluation Background" section presents some basic isolation-related terms and describes the evaluation criteria defined for this study. The "Software Attack Vectors" section presents the most common software attacks chosen to study security protection. The "MPU-Based Isolation Techniques" section presents an exhaustive description of the different chosen architectures. The "Evaluation" section evaluates and compares these architectures based on the defined evaluation criteria. The "Discussion of Possible Future Work" section discusses some possible future works to overcome some limitations of MPU-based architectures. The "Conclusion" section concludes the paper.

2 Evaluation Background

2.1 Terminology

In this section, we present some basic isolation mechanism-related terms used by the different architectures studied in this paper.

Lightweight Device A lightweight device is a low-cost device that does not need complex memory management. Therefore, it lacks an MMU and its memory model is referred to as a flat memory model because its memory appears as one contiguous address space. For example, devices with an Arm® Cortex®-M processor are lightweight.

Isolation By isolation we mean mechanisms that provide compartmentalization of software components. Compartments are separated and protected by a hardware component to prevent propagation of flaws from one compartment to the others. In our case, isolation is achieved with the MPU. This includes another layer of memory security by limiting compartments from accessing any memory address.

Memory Protection Unit A memory protection unit (MPU) is an optional hardware unit in some cores provided to protect memory. It allows only software components with high privileges, like a kernel, to define memory regions and attribute memory access permissions to each region. The MPU is used to restrict some memory regions to the software running under user mode. However, depending on the MPU version, there are some limitations. For example, for an ARMv7-MPU, which is present on some Arm Cortex-M devices as shown in Fig. 1, the base address

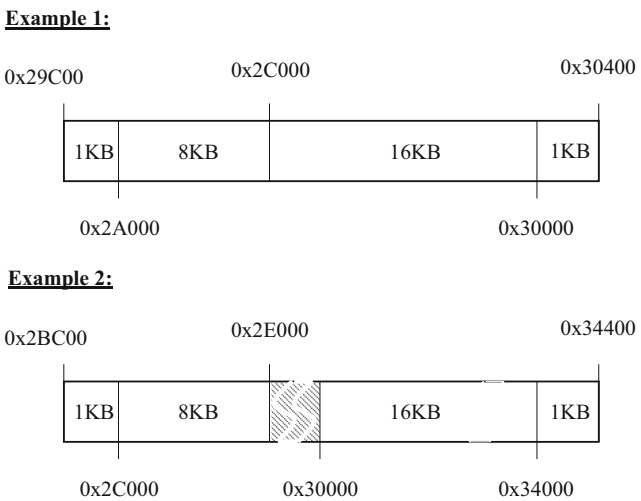


Fig. 1 MPU configuration: The MPU in the ARMv7-M architecture requires that the memory region size must be aligned to a start address. This address is a multiple of the region size, and the region size must be a power of two. In example 2, we can see the hole between the second and the third region

of a region must be aligned to its size, and the size must be a power of two. There is also another limitation, this time for all MPUs studied in this paper. MPUs only monitor memory accesses between the core and peripherals. Peripheral-to-peripheral accesses (e.g., *direct memory access* (DMA)) are out of reach to the MPU. Therefore, a compartment having access to the DMA can potentially read/write anywhere in memory.

Trusted Computing Base The trusted computing base (TCB) consists of a set of software and hardware components responsible for enforcing system protection and which are critical for device security; as examples, MMUs, MPUs, and hypervisors would be included in this category. The TCB design must be thought through carefully to guarantee a good security level. It should be kept as small as possible to reduce the surface that is exposed to attacks and to minimize bugs and weaknesses that could be used to break the system security.

2.2 Evaluation Criteria

This section defines the different criteria used to evaluate the selected isolation architectures. We divide evaluation criteria into four dimensions: *security features*, *security protection*, *TCB size*, and finally, *performance and memory consumption*.

Security Features The studied architectures propose different security features. To build a comparison between these architectures, we will preview all the mechanisms found in at least one architecture. These mechanisms are supposed to guarantee strong software isolation.

Inter-process communication Inter-process communication (IPC) refers to mechanisms provided by an operating system (OS) to processes so they can communicate with each other and share data. In trusted computing, IPC mechanisms must make communication between two processes secure and non-transparent.

Roots of Trust Roots of trust (RoTs) are a set of hardware and software components that are inherently trusted. They perform critical operations like measuring and verifying the software, protecting cryptographic keys, and performing device authentication.

Dynamic Application Loading Dynamic application loading is the ability to load, update, and delete applications at runtime.

Application Reboot Application reboot mechanism allows the device to reboot a specific application after being compromised, for example, without requiring rebooting of the whole system.

Exception Handling Exception handling mechanism makes sure that when an exception raises, it does not lead to any leakage of information.

Security Protection The strength of the protection of an isolation architecture is determined by the potential consequences of vector attacks. Admittedly, all architectures use an MPU, but the enforcement location [16] and the isolation granularity and the policies they enforce have an impact on their strength.

Trusted Computing Base Size The TCB is all the trusted components in a device that provide system security. The smaller the TCB, the smaller the attack surface. We will compare TCB sizes of all studied architectures.

Performance and Memory Consumption Lightweight embedded systems are very resource-constrained. Therefore, evaluating execution time and memory consumption represents a very important factor for developers and designers to consider in order to choose the right architecture.

3 Software Attack Vectors

We studied several vulnerabilities and exploits from different papers [17–20] and from the National Vulnerability Database (NVD) [15]. The NVD gathers vulnerabilities from the Common Weakness Enumeration (CWE) specification. It is a list of common security weaknesses that serve as a reference point for vulnerabilities identification, mitigation, and prevention effort. We also analyzed open issues and bugs in some architectures' GitHub repositories [8, 21] to identify other weaknesses.

For risk evaluation, vulnerabilities must be identified. And for a better identification, one can classify vulnerabilities. Our study of vulnerabilities and exploits shows that attackers can target different areas to compromise a system, we call these areas *security hotspots*. A *security hotspot* describes the sensitive areas where security is more critical than in other areas. This helps us to define what part of a device is targeted by a certain attack. It clarifies and helps to identify the best opportunities to compromise a device. The main security hotspots are *Authentication*, *Memory*, *Cryptography*, *Logic Errors*, *Synchronization and Timing*, and *Validation*. We chose to focus on the *Memory hotspot*. We present some issues and vulnerabilities in this hotspot. These can be divided into two main categories: *temporal errors* and *spatial errors*.

3.1 Temporal Errors

This category concerns attacks that take advantage of allocating, freeing, and deleting memory chunks. For example, in this category, there are *user-after-free* and *double free*

vulnerabilities. If exploited, they can lead to information leakage or to control-flow hijacking or even crash the program.

Use-After-Free A use-after-free vulnerability is when a program keeps using a pointer to a memory chunk that is not allocated anymore, and possibly re-allocated by another part of the program. The exploitation of this vulnerability can go from no effect to the execution of an arbitrary code.

Double Free A double free leads to an undefined behavior. This vulnerability happens when the same memory block is freed twice. Double frees can occur in different cases. For example, when two or more pointers point to the same memory block and begin cleaning using free(). The developer, if not careful, might free the same pointer many times. This might cause for other existing memory spaces to get corrupted or to fail future allocations.

3.2 Spatial Errors

This category focuses on spatial errors. Attackers can exploit spatial errors to execute code, to read/write the stack, or to halt the system. This can lead to unwanted behavior, and could extract sensitive data or change the defined program flow. In this category, vulnerabilities like *buffer overflows*, *stack overflows*, *heap overflows*, *format string*, *truncation*, and *signed convention* can be found.

Buffer Overflow A buffer is a memory area fixed to contain data. A buffer overflow occurs when the data written into a buffer overruns the buffer boundaries. And because of the contiguous memory space in lightweight devices, if the boundaries are not checked properly before writing or reading a buffer, it overwrites neighboring memory locations.

Most software developers have no security background; they can create unsafe code that leads to a buffer overflow. For example, the following code shows an example of an unsafe buffer copy that causes a buffer overflow:

```
char buffer[3] = "AAA";
char x = 'a';

strcpy(buffer, (char *)"TRUSTED");
```

Figure 2 illustrates a bit of the memory after we finish copying data into the buffer. The value of the variable x is overwritten. The results of exploiting a buffer overflow vary depending on the location of the overflowed buffer within the memory.

Stack-Based Buffer Overflow Stack-based buffer overflow is a buffer overflow where the program allocates the overflowed buffer on the stack. Attackers can exploit

Before strcpy:

A	A	A	\0	a

After strcpy:

T	R	U	S	T	E	D	\0

Fig. 2 Simple buffer overflow example

stack-based buffer overflows to manipulate the control flow of a program. When the system calls a function, a stack frame saves information related to this function. Then the stack frame contains information like the function parameters, local variables, and the return address. When a buffer stored in the stack is overwritten, the attacker can change data within the stack frame, especially the address stored within the return address. This allows the attacker to gain control over the execution path of the program. The following function shows an example of how an attacker can change the return address:

```
void function(void)
{
    char buffer1[8];
    char buffer2[16] = {0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x10, 0x00, 0x14, 0xab};

    strcpy(buffer1, buffer2);
}

// some routine executed

function();

// rest of the routine
```

Before explaining what occurs when the function is executed, we explain the stack manipulation when a function is called in Arm (Fig. 3 illustrates a stack frame):

- Push function arguments if there are any.
- Push the return address after the execution of function is finished.
- Allocate space for local variables.

The stack grows from higher memory to lower memory (Fig. 3), and the strcpy() starts copying data from a base address towards higher addresses. So, as Fig. 4 shows,

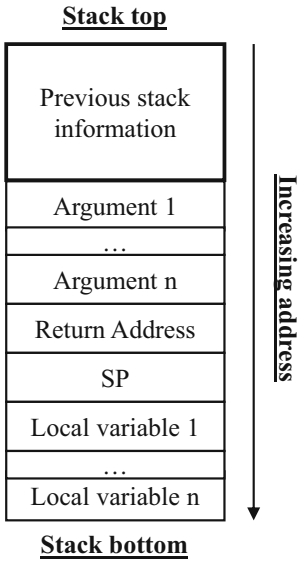


Fig. 3 Call stack operation

when buffer2, which has enough memory space for 16 bytes, is copied into buffer1 which has a memory area for only 8 bytes, the overflowed 8 bytes will overwrite data on the stack, and the attacker can change the value of the return address and hijack the execution path.

With a stack buffer overflow, attackers can inject code into the stack and redirect the path into this code. But some OSes provide defenses against code execution on writeable sections. However, many techniques have been developed to overcome these defenses. Return-oriented programming (ROP) is one of them; the attacker tries to reuse functions and gadgets. It is difficult to stop these attacks because the attacker executes valid existing codes or sequences.

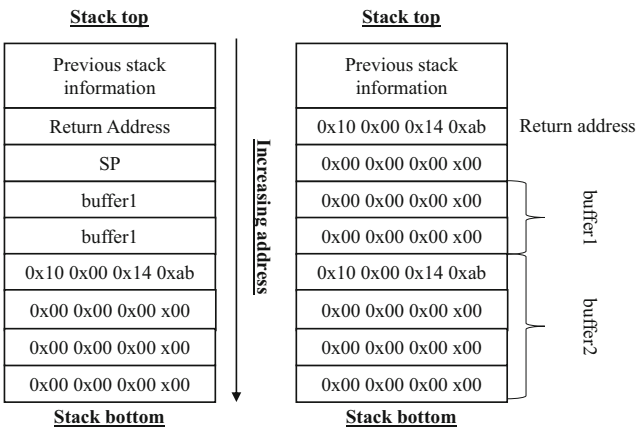


Fig. 4 Stack-based buffer overflow: On the left, a snippet of the stack before executing the `strcpy()`. And on the right, a snippet of the stack after executing the `strcpy()`. The buffer overflow leads to a change in the return address. Therefore, the path has been hijacked and when the subroutine finishes its execution, instead of returning to the mother routine, it will jump to the address 0x100014ab

Lower Boundary Check Usually, one side of the boundaries is checked, and it is the upper one. What about the other one?

The following piece of code presents an example of how not checking both boundaries can lead to exploitation:

```
char read_char(char * buff, int
buff_length, int index)
{
    if(index < buff_length)
        return buff[index];

    return -1;
}
```

If the attacker somehow controls the value of the parameter index, they could load index with a negative value and read 8 bytes from the memory. The function read_char only checks that the given entry index is smaller than the given buffer length and does not check the sign of index or its lower limit.

Format String The format string vulnerability occurs when there is a mismatch between the format string parameters (like %s %x) and the arguments of the format string function (like, for instance, print()). Exploitation of this weakness could lead, for example, to executing a code or reading the stack. The format string function does not have a limitation of entries. For each format string parameter, the format function will need an argument. If there are more format string parameters than arguments, the function will keep fetching data that does not exist in this function call stack. The following example shows how this vulnerability may crash a program:

```
printf("%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s");
```

The printf() will fetch for each %s a data from an address within the memory until a string termination character is found. There is the possibility that the address from which the printf() will try to read is an illegal address. It results in crashing the system.

An attacker can also read the stack, for example, in the following code, the printf() function will retrieve five parameters from the stack and print them on the screen:

```
printf("%x %x %x %x %x");
```

Truncation Truncation happens during conversion from a larger type to a smaller one. Data is usually lost, and sometimes it can lead to a manipulation of a branch condition, just like the following example shows:

```
void my_function(void)
{
    unsigned short size_temp;
    unsigned char * temp;
    unsigned char buff[512];

    temp = get_username();
    size_temp = strlen(temp);

    if( size_temp > sizeof(buff)) {
        error(0);
    }

    strcpy(buff, temp);
}
```

The variable size_temp is an unsigned short; it is included in the range [0, 65535]. If get_username() returns a buffer with 65900, size_temp will have the value of 364, which is smaller than sizeof(buff). This results in a *buffer overflow*.

Signed Convention Most *Type* weaknesses are consequences of conversions between signed and unsigned versions of the same type. When, for example, a signed integer is converted to an unsigned integer or vice versa, the value can change. The signed integers go from −2,147,483,648 to 2,147,483,647, while the unsigned integers go from 0 to 4,294,967,295. When a conversion occurs, the code checks the most significant bit to decide whether the value will be positive or negative. Consider the following code:

```
void read_data(unsigned int cursor,
unsigned char * buffer,
unsigned int length);

void my_function(int cursor, int length)
{
    unsigned char buffer[1024];

    // some routine

    if(length > 1024) {
        error(0);
    }

    read_data(cursor, buffer, length);

    // some routine
}
```

In this example, read_data() takes length as an unsigned integer, and my_function() takes length as a signed integer. If an attacker feeds my_function() with a negative value of length, the length check can be bypassed. But when read_data() is called, a conversion from signed to unsigned occurs, and the value will turn into a large unsigned value.

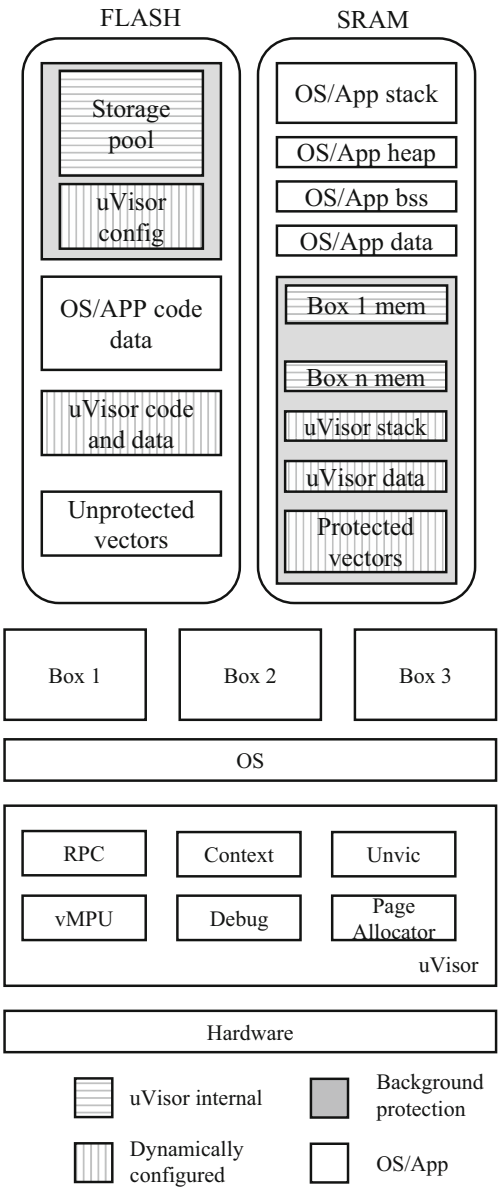


Fig. 5 uVisor architecture

In this section, most common vulnerabilities and how they could be exploited have been presented. In the following, these vulnerabilities will be used to compare security properties of MPU-based isolation techniques under study.

4 MPU-Based Isolation Techniques

We present in this section a detailed description of the studied architectures. Our architecture selection process was founded on solutions based on an MPU and designed for lightweight devices. We studied several solutions, industrial and academic, and we chose to present five of the widest spread and the most documented ones to this date. The chosen architectures

are uVisor [8], Tock [21], TyTAN [10], TrustLite [13], and ACES [22]. All these solutions are based on an MPU to protect the memory and provide compartmentalization.

4.1 uVisor

uVisor is an architecture for lightweight devices, an open-source software hypervisor which creates secure, separate domains called boxes, on Arm Cortex-M3/M4/M23/M33 microcontrollers, compatible only with Arm Mbed™ OS. The main objective of uVisor is to provide strong isolation between boxes and protect sensitive data from being leaked. Its TCB is composed of both software and hardware components.

uVisor is basically based on an Arm MPU, a trusted hardware component in the TCB. It provides compartmentalization by protecting each box memory area and restricting access to peripherals. The MPU guarantees that only authorized boxes can access a specific memory region. In case of a non-authorized access, a memory fault is generated.

A box represents a process, and for each one, uVisor allocates a heap and a stack and a static stack (see Fig. 5). It stores within the static stack the box context (sensitive global data only accessible from within the box). For each box, there is a constant access-control list (ACL) where authorized hardware peripherals and memories are defined for the box.

The TCB is composed of several secure software components. The virtual MPU (vMPU) configures the MPU upon box switch. An Arm MPU has only eight configurable regions. Therefore, thanks to the vMPU, we can declare more than eight regions for each box. When the OS switches to a box, uVisor reconfigures the MPU with the first eight regions from the ACL. Then, whenever there is an attempt to access a region that was unconfigured, the MPU will generate an error. The vMPU will retrieve the fault address and check within the box ACL if there is any region that contains that address. If it is the case, uVisor reconfigures the MPU with the right region and recovers from the fault. If it is not, this is clearly a non-authorized access.

uVisor provides a debug component to help users to identify problems and failures encountered during application execution. The user can create a debug box and customize system reaction to faults.

The remote procedure call (RPC) component allows a box to call functions that need to be executed in the context of another box. By default, boxes are unauthorized to call other boxes' functions that compute sensitive data. However, uVisor can declare RPC gateways to permit some functions to be called by other boxes. There are two types of gateways: synchronous and asynchronous.

The Page Allocator distributes pages and protects access to them. During uVisor initialization, it allocates heap memory into memory pages with the same size. A box can request pages from the Page Allocator, and if they are available, the Page Allocator secures the access for the box. Then the box can securely allocate memory inside the pages.

The Unvic component manages interrupts. Interrupts registration is exclusive to only one box. Registration is based on first-come, first-served basis. Other boxes can use an interrupt when it is freed by its owner. When interrupts occur, uVisor protects from data leakage between two domains by saving and clearing all important registers. Then it forwards them to their unprivileged handlers. These handlers are executed in the context of the owner box.

The context component is responsible for the context switch. This component is called by the OS, or by the Unvic component, or by the RPC one. This module calls the vMPU to reconfigure the MPU with the right configuration. In case it is called by the Unvic or RPC components, it also stores the stack state of the source box and sets the state of the destination box.

4.2 Tock OS

Tock OS in an open-source OS dedicated to Arm Cortex-M3/M4 microcontrollers. It achieves isolation and memory protection and is compatible with different languages. Tock is written in Rust [23], a programming language like C++ that provides better memory safety (no buffer overflow, no double frees). Its memory efficiency and performance are close to those of C++. Tock is also based on the Arm MPU to provide compartmentalization and memory protection.

Tock's kernel (Fig. 6, bottom side) is isolated from applications. The kernel is composed of two parts: a trusted one called the core that is responsible for critical tasks of the OS like scheduling and a non-trusted one called capsules that

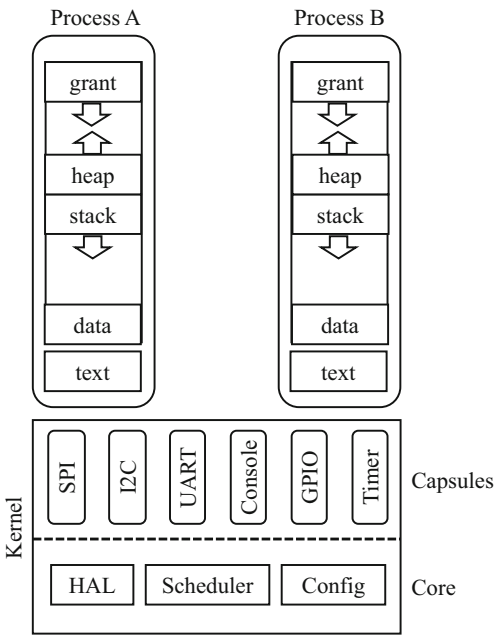


Fig. 6 Tock OS architecture

contains peripheral drivers and non-system critical tasks. Although capsules are non-trusted, they are running in the privileged mode and can communicate with the core and with each other to achieve their task. Rust's type system guarantees isolation between capsules.

Applications are processes (Fig. 6, upper side), they are untrusted, and they run unprivileged. They are isolated from each other, and each one has its own memory region protected by the MPU from non-authorized accesses like, for example, from other processes. If a process crashes, it does not make the system crash; it can restart without interfering with the kernel. It is also possible to load processes at runtime without restarting the kernel. Processes can communicate with each other via inter-process communication (IPC), and with the kernel through system calls.

Although processes can be written in any programming language that supports position-independent code (PIC) and the Cortex-M architecture, C and C++ are the only languages that are widespread in writing such applications.

System calls allow unprivileged applications to communicate with the kernel. Tock provides 5 system calls: command, allow, subscribe, memop, and yield.

The command system call tells capsules to execute a specific action synchronously. It takes four word-sized entries. The first entry tells the kernel which capsule should run the task. The second one specifies the requested command. The third one provides more fine-grained actions, and the last one contains the caller identifier. Command should not be long to execute; however, commands can start an asynchronous task via a subscribe system call.

The allow system call is like the command system call, but used when there is a large buffer to transfer. It defines a memory region as shared between a capsule and an application. It takes four arguments: The first one specifies which capsule will be granted access. The second one determines the purpose of the call. The third and the fourth ones take a pointer to the start address and its size, respectively.

The subscribe system call sets up callback functions for capsules to be run in response to an event. It takes as arguments the capsule number and a callback function pointer. Once the event is triggered, the callback is fired.

The memop system call invokes the core to expand the memory available to the process.

The yield system call pauses process execution until the callback completes.

Tock also provides a kernel abstraction called *grants*. They allow capsules to have access to an allocated memory region within each process. This memory region is not accessible when a process dies. Capsules will need to allocate memory to execute processes demands, as such demands cannot be anticipated in advance. So, rather than allocating resources statically and wasting lots of memory, grants solve the problem as capsules can dynamically allocate memory from the grant area within each process, and the grant area cohabits with the process heap area.

4.3 TyTAN

TyTAN is a security architecture for small devices. It is based on an Intel's Siskiyou Peak architecture, and on modified FreeRTOS. It provides dynamic loading and configuration of secure tasks, secure IPC, and real-time guarantees.

Figure 7 presents the architecture of TyTAN and its different components. There are two different types of tasks: Secure tasks and normal tasks. Normal tasks are isolated from secure tasks. Secure tasks are isolated from each other and from other software components. Their memory regions are protected using the execution-aware MPU (EA-MPU). Tasks can be loaded dynamically using an ELF loader.

The TCB is composed of many parts, such as a secure boot, a Root of Trust Measurement (RTM) task, an EA-MPU driver, etc. The secure boot is responsible for loading all other parts of the TCB to ensure their integrity. The RTM task is used for the tasks' integrity: it computes a cryptographic hash of the binary task code that represents the identity of the task. This calculus is the basis for local and remote attestation. For local attestation, task identity can be used to attest the task. For remote attestation, the Remote Attest task uses a message

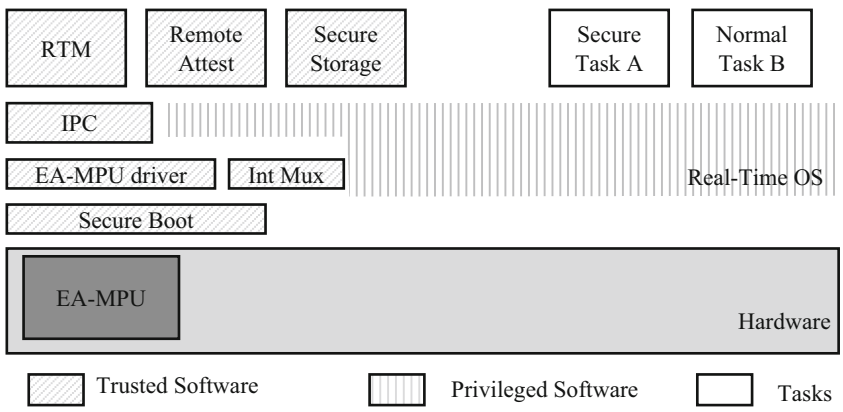


Fig. 7 TyTAN architecture

authentication code (MAC) with an attestation key to prove the authenticity of a task identity to a remote verifier.

The EA-MPU driver is responsible for configuring the EA-MPU dynamically at each load or unload of application. The isolation is based on the program counter (PC). The EA-MPU driver configures the eighteen slots with the memory regions and their access control rules.

Tasks communicate with each other using an IPC proxy task. This one allows transmitting a message *m* from a sender task *S* to a receiver task *R*. For small messages, *S* copies *m* and *idR* (identity of task R) into the CPU registers and calls the IPC task via a software interrupt (SWI). The IPC task using *idR* determines *R* memory region and writes *m* and *idS*. For large messages, the IPC task creates a shared memory region between the involved parts.

TyTAN offers a secure storage where tasks can store their sensitive data. It is based on data encryption. For each task, a key is derived from its identity. The key is used to encrypt data before storing it into the secure memory. The uniqueness of the task identity guarantees the uniqueness of the encryption/decryption key.

TyTAN also offers an interrupt multiplexer task used to securely save task context and clear central processing unit (CPU) registers before the interrupt handler takes control. There is an interrupt descriptor table (IDT) protected by the EA-MPU where handlers for interrupts are determined.

The main objective of TyTAN is to ensure the integrity of critical tasks while maintaining the real-time criteria in low-power microcontrollers. This is done through the secure boot and the EA-MPU.

4.4 TrustLite

Just like TyTAN, TrustLite is based on Intel's Siskiyou Peak architecture and uses a very modified EA-MPU. Unlike TyTAN, it is a generic protected module architecture (PMA), OS-independent. TrustLite provides compartmentalization and guarantees code and data integrity and confidentiality of the compartments.

TrustLite compartments are called *trustlets* (see Fig. 8). A trustlet can contain many software components. It is defined by its code, data, and other memory regions and an entry vector. This entry vector contains one or more entry points

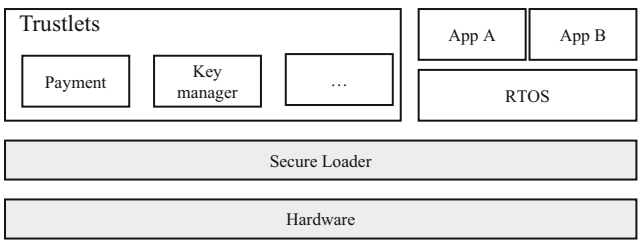


Fig. 8 TrustLite architecture

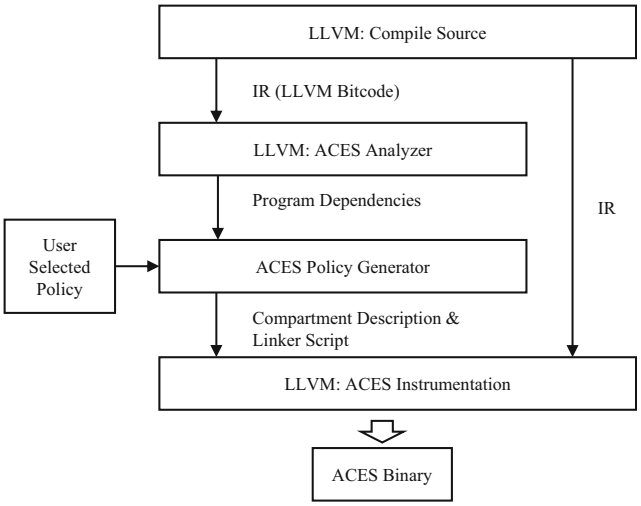


Fig. 9 ACES tool overview

that allow other tasks and trustlets to execute authorized functions of the corresponding trustlet.

The MPU has 32 protection regions, and for each trustlet, we declare multiple different regions. These memory regions are only accessible by the corresponding trustlet. The MPU gives a trustlet access or not to a memory region depending on the PC, just like TyTAN. If the PC is inside the trustlet code, it has access to the authorized memory regions; if not, an access violation exception is raised.

TrustLite also handles interrupts. Its interrupt manager is called the Exception engine. To ensure that no information is leaked while interrupting a trustlet at any time, the Exception engine stores the CPU state to the current trustlet stack, stores the stack pointer in the trustlet table, and clears all general-purpose registers. After executing the interrupt handler, the interrupted trustlet resumes running by jumping to its entry point and restoring its stack as quickly as possible.

TrustLite provides a protected IPC and an unprotected IPC. In TrustLite, signaling and short messages are done in a form of an RPC. Concerning large buffers, trustlets use a shared memory defined in advance in an MPU region. The size of the page and the participants are defined. Protected IPC uses a simple handshake protocol to attest the identity of the receiver and sender, and to create a cryptographic session key used to authenticate messages in both directions.

4.5 ACES

Recently, research has been proposing a different kind of compartmentalization solutions. Such solutions [22, 24] are LLVM-based embedded compilers and offer an automated way to create two or more isolated execution environments. While EPOXY [24] offers two compartment solution based on privileges, ACES (Automated Compartments for Embedded Systems) [22] offers the opportunity to have multiple compartments in bare metal applications. This paper focuses on ACES.

ACES is an LLVM-based compiler that creates automatically isolated environments in bare metal applications. ACES fulfills its objective by performing four steps (see Fig. 9): Program analysis, compartment generation, program instrumentation, and run-time enforcement. The program analysis (step 1) creates the PDG (program dependence graph), and it captures all control-flow, variables, and peripheral dependencies. This is used to generate the region graph. The region graph is then used to generate compartments (step 2).

The user provides a compartmentalization policy; there are 3 policies: Naïve Filename, Optimized Filename, and Peripheral. The Naïve Filename consists in considering each file name a compartment. The Optimized Filename consists in merging files with same region access into the same compartment. And the Peripheral policy consists in grouping code that accesses a single peripheral into the same compartment. Then ACES applies the chosen policy.

To meet the 8 region MPU constraint, ACES merges some regions with each other. The program instrumentation (step3) creates a compartmentalized binary. It instruments the program and identifies all compartment transitions. All cross-calls are identified, and it creates a metadata. This metadata is used to validate a transition.

After instrumenting the binary, ACES lays out the program in memory to meet the MPU constraints which are region sizes are powers of two and the region start address is aligned to its size. This routine is done in two steps: In the first round, ACES uses a linker script that ignores these constraints; the resulting binary is used to extract region sizes. Then, in the second round, ACES uses another linker script to expand memory regions into power of twos and lays out regions from the highest to smallest to minimize holes between regions. The resulting binary is ready for execution.

5 Evaluation

After presenting a detailed description of the five studied MPU-based architectures, in this section, we will evaluate and compare the solutions based on the evaluation criteria defined in the "Evaluation Background" section. Comparisons that are non-relevant are not included.

5.1 Security Features

As seen in Table 1, all architectures provide the IPC mechanisms. uVisor provides two types of IPCs. The first one is in a form of RPCs to execute functions in the context of other compartments. Every RPC can take up to four arguments of 32 bits. The second one allows exchange of large buffers. TyTAN has a task dedicated for exchanging messages between two processes via a software interrupt, and for large buffers, TyTAN allocates a shared memory between the two involved parts. TrustLite is similar to TyTAN. Furthermore, it also proposes RPCs. Only TyTAN and TrustLite verify the identity of the involved parts before exchanging messages. For uVisor, developers can create their own mechanism to verify the identity of the caller, and each compartment can be given a name. However, uVisor does not check for those names' uniqueness. Therefore, compartments can have the same name, and this can lead to a confusion identifying the right one.

Only TyTAN provides remote attestation. The RTM task in TyTAN is responsible for computing a cryptographic hash, which is the identity of a compartment. This calculus is used for local and remote attestation.

Both Tock OS and TyTAN provide dynamic application loading, and both use position-independent code (PIC) to enable this feature. This mechanism allows developers to load applications into any memory address at runtime. The code can be placed at any address because all branches are PC relative rather than absolute.

Concerning application reboot in the case of an exception, in Tock OS, the application is rebooted without interfering with the kernel. But in uVisor, it provides a debug box that retrieves exception information with some tools, and it is the developer who should add reboot compatibility. As of today, only device reboot is possible in uVisor. Application reboot is planned for next releases.

All architectures handle exceptions in similar ways. They save then clean all important registers.

5.2 Security Protection

Note that isolation architectures are not aimed at protecting against specific attacks. Rather, they provide

Table 1 Summary of security features studied in this paper

	Inter-process communication	Root of trust	Dynamic loading	Exception handling	Application reboot
uVisor	●	○	○	●	○
Tock OS	●	○	●	●	●
TyTAN	●	●	●	●	○
TrustLite	●	○	○	●	○

●, yes; ○, no

compartmentalization to prevent flaws and bugs in one software compartment to propagate to other compartments of the device.

The first difference to observe between the five architectures is that except for Tock OS, all the others are written in C, which is an unsafe language. Tock OS is written in Rust, a memory-safe language. Rust is supposed to deal with out-of-bounds pointers and with dangling pointers that are the sources of most memory attacks. So how Rust achieves memory safety? Rust relies on three features: ownership, borrowing, and bounds-checking. For ownership, the compiler tracks the ownerships of each value. A value can only be used once and, then, the compiler refuses to use it again. This way, Rust prevents double-free errors regularly found in C/C++ language. Concerning borrowing, since Rust has rules about having one mutable pointer to a variable at a time, it employs borrowing to offer developers the possibility to pass references. References are immutable by default. However, we can have mutable references. Mutable references are possible only if we have one mutable reference to one variable in a particular scope. This way, Rust can prevent data races at compile time. Finally, Rust offers bounds-checking, but it comes at a cost. Except for C, other languages have support of it.

However, when developers use the *unsafe* keyword, they write their code in other languages like C, for example, and those parts are prone to memory attacks. A developer can reproduce every security vulnerability found in C. And if an attacker succeeds in hijacking the *unsafe* code, the safe one can be hijacked as Song shows in his paper [18]. To date, the Tock OS kernel contains *unsafe* code. Also, Rust language libraries and mechanisms are not totally written in Rust. The interface is safe, but the underlying implementation is written in an *unsafe* code. It is also important to note that to date, it is impossible to write applications in Rust [25]. This means applications in Tock OS are written in C/C++.

We classified vulnerabilities in the "Software Attack Vectors" section into two categories: *temporal* and *spatial* vulnerabilities. An attacker can modify code, data variable, or code pointer. This can lead to a code corruption attack, a control-flow hijack, or a data value attack [17]. For code corruption attacks, the attacker tries to modify the code into their own. But the five architectures define text sections with the ReadOnly attribute, so they prevent attackers from overwriting the code.

In the following, let us assume a system with two different compartments, A and B. The code of A is unsafe and contains many weaknesses, while B is safely written because it processes some sensitive data.

For data value attacks, the attacker tries to modify data values to change the execution path. If the attacker can exploit a buffer overflow from within compartment A and tries to reach a data value within the compartment B memory region,

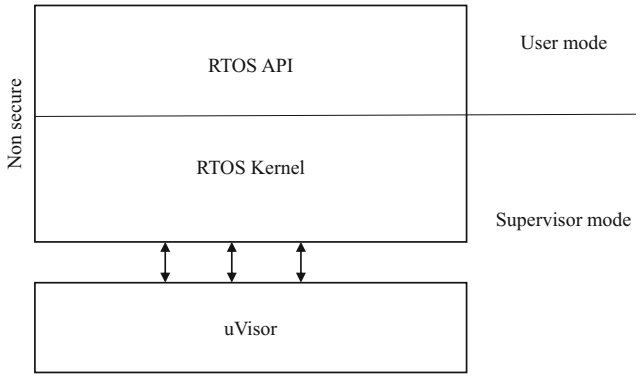


Fig. 10. uVisor and RTOS have the same privileges: RTOS is non-secure, and contains many buffers, but it has the same privileges as uVisor, which can be critical for system protection

the MPU will raise an exception. The attacker can only read/write memory regions with the MPU R/W attributes; they can manipulate authorized memory regions. And while the MPU guarantees protection of compartments' memory regions, attackers cannot access, from a compartment, other compartments' resources.

For control-flow hijack, an attacker will try to redirect a code pointer to the address of their injected code, or to an already existing function or gadget. This is usually done by manipulating the return address after exploiting, for example, a *stack-buffer overflow* (see the "Software Attack Vectors" section) or by using indirect jumps or call instructions. None of the architectures mark their stack and heap segments as non-executable, by default, to prevent executing injected code. So, if an attacker succeeds in exploiting a buffer overflow or a dangling pointer, they can inject their code and redirect the path to their own code.

Except for ACES, developers have to change stack and heap attributes into *non-executable* in the other four architectures' code to prevent executing injected code. But even if they do so, attackers still can reuse existing functions and gadgets. The code can be divided into two categories: shared libraries' code and compartments' code. For the shared libraries' code, like *libc*, for example, the code is publicly accessible. Reusing their functions will not be prevented, but if an attacker tries to process data from a protected compartment, the MPU will raise an exception. The functions will be executed, but only in the context of the running compartment.

Tock OS, ACES, and TyTAN protect a compartment code from being executed in the context of other compartments [9, 10, 22]. In the case of TrustLite, the developers must choose if they want to authorize a compartment to execute another compartment code or not. uVisor does not provide such a protection. An attacker can call any function, but, just like shared libraries, these functions will be executed in the running context, which means that if the function processes some resources that are not authorized for the running compartment, the MPU will raise an exception.

Table 2 Summary of security protection studied in this paper

	Temporal attacks	Spatial attacks	Code read-only	Code isolation	RAM no-execute
uVisor	○	○	●	○	○
Tock OS	◒	◒	●	●	○
TyTAN	○	○	●	●	○
TrustLite	○	○	●	●	○
ACES	○	○	●	●	●

●, yes; ○, no; ◒, partial

Except for ACES, within all the other architectures, the OSes have the same privileges as the TCB. Tock OS uses a new OS written in a safe language, while the others use traditional OSes that were developed a long time ago, did not take security measures, and use lots of buffers. This increases the chances of buffer overflows within a privileged component, and it could be critical for system security, as we will demonstrate for uVisor (see Fig. 10).

First, note that in mbed RTOS, the svc 0 is reserved for RTOS SVC Handler and it is used by the OS to switch from user mode to a privileged one. If we have a look at the **SVC Handler** function, we will find that the RTOS trusts the register R12, and it contains the address of the function to execute in privileged mode.

The SVC Handler can be called from any compartment by simply executing svc 0. Now, we take the example of the other part of the system with two compartments, A and B. In uVisor, if we have a look at how the memory is configured in the MPU, there is no protection against execution of injected code. So, from a vulnerability within A, we can inject the following simplified code:

```
LDR R12,=exploit
SVC 0

exploit:
...
```

This will load R12 with the exploit address and then call the SVC Handler. The handler will branch to the exploit in privileged mode and the game is over. We have succeeded in disabling the MPU, for example, and running the system without protection.

Table 2 summarizes security protections studied in this paper. Except for Tock OS, which is partially written in a memory-safe language and protects partially from memory corruption attacks, the others are not protection solutions. But, all five architectures limit the consequences of such attacks by preventing memory regions from being publicly accessible. Moreover, running a privileged OS can be fatal. uVisor does not trust mbed OS, but they are still running with the same privileges. As shown above, an attacker can take advantage of the OS and escalate their privileges.

5.3 Trusted Computing Base

This section will compare the TCB size of the studied architectures. It is more appropriate to compare between uVisor and Tock OS and ACES, and between TyTAN and TrustLite. The first reason for this is because uVisor and Tock OS and ACES and TyTAN and TrustLite are not based on the same cores. Secondly, TyTAN and TrustLite are not open source and there is not enough available information about the software components.

uVisor, Tock OS, and ACES use practically the same MPU (Armv7-MPU). On one hand, uVisor has only 7kLOC (line of code), meanwhile Tock OS has 11kLOC. However, while uVisor is just a hypervisor and needs an OS to work, uVisor is only compatible with mbed RTOS. mbed RTOS is not trusted, but is executed in privileged mode. uVisor still provides more flexibility than Tock OS and is also compatible with three different MPUs (ARMv7m, ARMv8m, and Kinetis). On the other hand, ACES has only 1.3kLOC. This is since ACES is destined to bare metal applications, there is no OS. So, there is no security features like the other architectures. ACES TCB manages MPU reconfigurations and system calls.

TyTAN and TrustLite are based on the EA-MPU. For TrustLite, designers took the OpenMSP430 and radically changed the EA-MPU. As we explained in the "TrustLite" section, the MPU is configured during boot time. It contains all trustlets and the OS configurations. Then, based on the PC, it authorizes or declines access to a memory address. Also, the MPU can contain 32 regions. The overhead in FPGA slices is around 200% [13] of an openMSP430. For TyTAN, designers took the same openMSP430 and TrustLite MPU, then they cut it down to support only 18 regions, which reduces the overhead compared with TrustLite to 100%.

To summarize this section, Tock's TCB is around 4kLOC bigger than uVisor's because Tock includes the OS in the TCB. TrustLite and TyTAN use a custom MPU. TrustLite's MPU is twice as large as TyTAN's because it supports more memory regions.

5.4 Performance and Memory Consumption

In this section, the performance and memory consumption of uVisor and Tock OS are compared, and when possible, ACES and TyTAN too. TyTAN's and TrustLite's codes are not open to

Table 3 Performance of creating a process (in clock cycles)

Routine	uVisor	Tock OS	TyTAN
Process creation	5500	2900	642,300

the public, so we could not perform our own evaluation tests. For comparison purposes, we used literature data when available. ACES is destined to bare metal applications; therefore, we cannot compare all performance sides, such as process creation. Tests were carried out on an Arm Cortex-M4 board with 2 MB of FLASH and 192 KB of RAM. uVisor and ACES were already ported to this board, but not Tock OS, so the minimum required in order to evaluate and compare the two architectures was ported by our own work.

5.4.1 Performance

The evaluation focused especially on some components in the kernel and hypervisor (for uVisor) side. We evaluate mechanisms like task creation, MPU configuration, the context switch, interrupts, and the cost of an MPU fault recovery for uVisor to add an authorized region that was not pre-configured.

Process Creation The process creation scheme differs from one architecture to another. For uVisor with mbed OS, it is done in two steps. In the first step, uVisor loads all boxes' static configuration from the FLASH and allocates memory for every box. The second step occurs during the OS start and consists in creating the corresponding processes. Tock OS and TyTAN both support dynamic application loading. Process creation consists of relocating the process, loading the corresponding MPU configuration, and finally, only for TyTAN, the authentication of the process. Table 3 presents the performance of creating a process in every architecture. TyTAN has the biggest runtime overhead and this is because TyTAN authenticates every process before loading it, and it takes around 433,400 clock cycles [10] to measure a process. uVisor does not authenticate processes, but it automatically allocates memory for processes. And because of the MPU (Armv7) limitations seen in the "Evaluation Background" section, more processing time is needed to load configuration processes: to round up memory sizes into power of twos, to shift the memory start address to be aligned with its size, and to load the process's access control list. Tock OS has the best performance of the three architectures because it does not authenticate applications. And applications have very small configurations—practically all configurations are static and the same.

Table 4 Performance of configuring the MPU (in clock cycles)

Routine	uVisor	Tock OS
MPU configuration	390–4450	560

Table 5 Performance of peripheral registers writing (in clock cycles)

Routine	uVisor	Tock OS	ACES
Register writing	6	340	6

MPU Configuration In Tock OS and uVisor, the MPU configuration is dynamic; therefore, when a thread-switch or an interrupt occurs, the MPU needs to be configured with the right rules. Table 4 shows the cost in clock cycles for such a routine. On one hand, Tock OS has only two regions to configure for each application: its text section and its data section, and the configuration is done in 560 cycles. On the other hand, uVisor has two or three static regions, depending on the board, which are the public regions in the SRAM and the FLASH for all applications. The other six regions are filled from the application ACL (access-control list) with the right rules. Depending on the ACL, the routine can take from 390 to 4450 cycles. The MPU configuration in Tock OS takes less time than in uVisor and this is because uVisor offers more flexibility in defining memory access rules. But, as we will see next, the non-flexibility of Tock OS can be very costly when an application needs to access different peripheral registers or other memory regions like the non-volatile SRAM, for example.

Writing in a Peripheral Register Embedded systems interact a lot with peripherals, and it is very rare to find an application that does not make use of the system peripherals. Because of the very limited access of applications in Tock OS, we thought it would be interesting to compare between uVisor, ACES, and Tock OS accessing a peripheral register. Table 5 presents the results of this test. An application in uVisor and ACES needs less time than in Tock OS to write into a register because it was already authorized (in the ACL) to access that memory region. In Tock OS, an application can access only its data and text sections, so to write in a peripheral register, the developer needs to define a capsule for the corresponding peripheral. Capsules are part of the kernel and are executed in privileged mode. Calling a capsule from an application requires a supervisor call (SVC), then the kernel redirects to the right capsule which writes into the register, and finally, execution mode is changed to user mode and the system returns to the application. It costs 340 cycles for a simple memory access.

Interrupts For interrupt handling, the test consists in evaluating the switch in the interrupt handler and the switch out to

Table 6 Performance of interrupt switch in and switch out (in clock cycles)

Routine	uVisor	Tock OS
Interrupt	290–4740	3160

Table 7 Performance of context switching (in clock cycles)

Routine	uVisor	Tock OS	ACES
Context switch	2040– 6100	810	675

resume the execution flow. Table 6 shows the results for the two architectures. For uVisor, it varies from 290 to 4740 cycles because of the MPU configuration. There is a use case where switching in and out needs only 290 cycles. It is the case when the interrupt occurs during the process owning the interruption handler (see the "uVisor" section). Here, uVisor performs ways better than Tock OS. In Tock OS, it takes 3160 cycles, because of the MPU configuration. There is also a high latency because the interrupt handler cannot be invoked directly; the kernel has to find the right handler to call.

Context Switch For the context switch, Table 7 shows that in ACES, it takes 675 cycles to switch from a context to another. Tock OS takes 810 cycles, while uVisor takes from 2040 to 6100 cycles. This huge difference is, again, due to the MPU configuration. There are up to six more regions that can be configured during this step, and during a process switch, the RPC mechanism needs to drain outgoing RPC queues (see the "uVisor" section).

uVisor MPU Recovery Mechanism This mechanism shows how a process can configure up to 16 memory regions while the MPU has only eight memory regions. In uVisor, two slots are reserved for the public RAM and FLASH, and another slot is reserved for the process heap and stack. So, developers have only five slots to configure. To overcome this limitation, an MPU recovery mechanism allows developers to set rules for more than five memory regions. Then, during the MPU configuration, the first five memory regions in the ACL are configured in the MPU. If the application tries to access an authorized address that is not configured in the MPU, the MPU will raise an exception. uVisor retrieves the fault address and checks if it is within an authorized region. Here, uVisor overwrites a memory region with less or the same priority in the MPU and recovers from the fault. As shown in Table 8, this step costs 560 cycles, but it allows an application to have access to more than eight memory regions.

5.4.2 Memory Consumption

The memory consumption is divided into two parts. The first part is the memory used when no task is loaded. Table 9 shows that uVisor consumes less FLASH than Tock OS and TyTAN.

Table 8 Performance of uVisor MPU recovery mechanism (in clock cycles)

Routine	uVisor
MPU recovery	550

Table 9 FLASH memory consumption (in kB)

uVisor	Tock OS	TyTAN
68.27	72.58	244.08

The second part is the memory consumption within the RAM and how uVisor and Tock OS manage RAM memory. Table 10 presents the initial RAM memory consumption. uVisor alone needs only 2 kB, but with mbed OS, 8.60 kB are needed. Tock OS needs 7 kB. The results are very close. The difference between both architectures remains in managing memory for processes. uVisor and Tock OS have two different policies. uVisor has a dynamic policy where developers define the amount of memory for a process, then during uVisor boot, it is rounded to match MPU limitations (see the "Evaluation Background" section). In Tock OS, the amount of memory needed for all processes is the same, and it is fixed within the kernel by the developer. Both policies have their advantages and disadvantages. In uVisor, developers decide if they need a big chunk of memory or just a small one, depending on the application needs. In contrast, in Tock OS, if there are applications with variable memory block needs, they will have the same amount. This can lead to lots of memory waste, especially when one particular application needs lots of memory space, while others do not need too much. But, in uVisor, the fact that memory regions are not fixed can lead to another problem. The ARMv7 MPU limitation can lead to some waste in memory (see the "Evaluation Background" section). To limit the waste, uVisor has a routine to order memory regions of applications. These limitations were fixed for the ARMv8 MPU.

6 Discussion of Possible Future Work

Table 11 summarizes some important results of the studied MPU-based isolation architectures. All the four architectures have their advantages and disadvantages. Except for uVisor, each architecture is suited to a specific type of applications. uVisor flexibility and small memory consumption make it suited to a large panel of applications. However, the results from Table 11 show that the deterministic level can be lost if applications get too complicated. As seen in the "Evaluation" section, much effort has been directed towards these architectures, but there is still room for enhancement. For example, in uVisor, root of trust and dynamic loading can be good assets. Tock OS is minimalist, and it is designed for tiny and simple embedded systems, such as those with a very limited number

Table 10 Initial RAM memory consumption (in kB)

uVisor	Tock OS
8.60	7

Table 11 Summary of all MPU-based isolation architectures studied in this paper

	Security features						Memory consumption			Performance (clock cycles)					
	Inter-process communication	Root of trust	Dynamic loading	Exception handling	Application reboot	Memory safety	TCB size (kLOC)	Flash consumption (kB)	RAM consumption (kB)	Process creation	MPU configuration	Register writing	Interrupts	Context switch	MPU recovery
uVisor	●	○	○	●	○	○	7	68.27	8.60	5500	390 4450	6	290 4740	2040 6100	550
Tock OS	●	○	●	●	●	◒	11	72.58	7	2900	560	340	3160	810	○
TyTAN	●	●	●	●	○	○	?	244.08	?	642,300	?	?	?	?	○
TrustLite	●	○	○	●	○	○	?	?	?	?	?	?	?	?	○
ACES	○	○	○	◒	○	○	1.3	–	–	–	–	6	?	675	○

●, yes; ○, no; ◒, partial; ?, NA; –, non-relevant

of processes and, especially, for applications that spend most of their time in sleep mode. But it does not mean Tock OS has less power consumption than other architectures. It has been shown [26] that Tock OS consumes more power than mbed OS, especially when an application requires a lot of computational work, as cryptographic operations do.

For both uVisor and Tock OS, runtime performances can be improved. For Tock OS, we noted that the MPU was configured at every system tick, even if the OS does not really switch to another thread. Also, during interrupts, sometimes reconfiguring the MPU is unnecessary and there is much time wasted in the kernel finding the right handler. For uVisor, providing developers with up to 16 regions to configure for each process is a good advantage compared with all other architectures. But it has its cost on performance as shown in the "Performance and Memory Consumption" section. Maybe the solution for uVisor will be to use a different MPU—one that will allow it to configure all memory regions at boot time. This would suppress the cost of the MPU configuration for every thread-switch and interrupt. But as seen in TrustLite, this might also have a prohibitive cost and might offer fewer memory regions to configure for developers.

Tock OS uses Rust in its kernel, which is a memory-safe language. But for low-level operations, Tock OS uses an unsafe language. As seen in the "Evaluation" section, an attacker can control the execution flow from an unsafe code. Applications in Tock OS cannot be written in Rust and, to date, they are all written in C/C++. But they are very limited and have access to only two regions. It would be better if they could be written in Rust, too, to lower the attack surface. The three other architectures are written in C and assembly. Unfortunately, only uVisor of the three is open source, so we could not review TyTAN and TrustLite code. uVisor is a small hypervisor and its TCB size is around 7kLOC, but it needs mbed OS to work. In the "Security Protection" section, we showed that there is a non-negligible part that runs in the privileged mode. We demonstrated how, from the user mode, we can call the OS and from then execute an arbitrary code in privileged mode. To overcome this, uVisor has to configure the RAM memory as non-executable. Then, to reduce the attack surface, it may need to run the OS with less privilege than uVisor. To do so, we can opt for *para-virtualization*. The timer (*Systick*) used by the RTOS could be handled by a para-virtualized one in non-privileged mode. Using para-virtualized hardware will for sure have an impact on performance, especially on interrupt latency. But determinism can be assured if the para-virtualization is done correctly. We could also reduce the time hogged by the MPU configuration at every thread-switch and every interrupt.

ACES is an automated solution but for bare metal applications only. ACES saves developers from the engineering needed in the other architectures to create isolated execution, allocate the right heaps and stacks, and defining MPU

memory regions. However, ACES does not deal with interrupt handlers. These are handled in privileged mode and attackers can take profit of a malicious handler to break system security. Furthermore, ACES can include on average 5% run time overhead [22] compared with uVisor. This because ACES offers a more fined compartmentalization granularity and its switch context is quite costly, although this fine granularity reduces ROP attacks on average by 94.3% [22].

Another issue in all these solutions is the direct memory access (DMA) protection. MPUs in the four architectures are interfaced with the CPU, which means only software memory accesses are controlled. And because the DMA is outside the CPU, it can access all memory regions bypassing all architectures' security measures. Overcoming DMA attacks is very challenging, especially from a software standpoint. Tock OS kernel has a software abstraction layer that partially limits the MPU circumvent. Rust memory safety ensures the kernel exposes memory-mapped registers safely, so applications cannot write arbitrary values beyond the boundaries [23]. A complete software solution may be difficult and costly in terms of performance. A software/hardware or just a hardware solution may be better. The first solution that came into our minds is to add an MPU that controls memory accesses of the DMA. It can be costly and not appealing for industrials. Another solution we can think of is to change the MPU position within the chip in a manner such that it controls both CPU and DMA.

To summarize this section, let us consider an architecture that proposes a higher level of control and a more favorable cost-performance ratio and more safety. An architecture, as the outcome of this paper, has to be able to detect attacks and recover, to limit OS privilege, and enhance memory protection and flexibility with an affordable cost. For such a solution, considerable work is needed within the actual MPUs and the software managing this MPU. Microsoft, for example, already started thinking about this subject. A group of researchers there proposed Sopris [27], a whole chip where they use an MMU instead of the classic Arm MPU for full isolation. But they did not consider the DMA issue. This is part of our future work and it is beyond the scope of this paper.

7 Conclusion

In this paper, a study of most known MPU-based isolation architectures was done. The study covered a comparison of isolation robustness and security features, as well as performance and memory consumption of some architectures (unfortunately, not all architectures are open, so we could not address more of them). This study shows that all architectures provide strong isolation, and they limit the consequences of software attacks, making it harder for an attacker to reach their goal. We evaluated the performance costs of MPU-based isolation, and the results of costs were consistent with the policies chosen in each architecture. Nevertheless, the study presented some weaknesses and limitations in every architecture and demonstrated how reusing a gadget from the OS can be fatal. Finally, we presented some ideas of possible works for better isolation and to reduce the runtime overhead.

Much work has been done in the design of lightweight devices in recent years; however, there are still many challenges for researchers in enforcing isolation and improving performance to yield more resilient systems.

References

1. Kate Temkin MS (2018) Fusee Gelee exploit
2. (2018) Shofel2 exploit
3. de Clercq R, Verbauwhede I (2017) A survey of hardware-based control flow integrity (CFI). CoRR abs/1706.07257:
4. 2017 I (2017) Intel control-flow enforcement technology preview
5. Qualcomm Technologies I (2017) Pointer authentication on ARMv8.3
6. Davi L, Hanreich M, Paul D, et al (2015) HAFIX: Hardware-Assisted Flow Integrity eXtension. 2015 52nd ACM/EDAC/IEEE Des Autom Conf 1–6
7. Karger PA, Schell RR (2002) Thirty years later: lessons from the multics security evaluation. ACSAC, In
8. ARM (2015) uVisor. GitHub Repos
9. Levy AA, Campbell B, Ghena B, et al (2017) Multiprogramming a 64kB computer safely and efficiently. In: SOSP
10. Brasser FF, Mahjoub B El, Sadeghi A-R, et al (2015) TyTAN: tiny trust anchor for tiny devices. 2015 52nd ACM/EDAC/IEEE Des Autom Conf 1–6
11. Noorman J, Agten P, Daniels W, et al (2013) Sancus: low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In: USENIX Security Symposium
12. eChronos (2018) eChronos
13. Koeberl P, Schulz S, Sadeghi A-R, Varadharajan V (2014) TrustLite: a security architecture for tiny embedded devices. EuroSys, In
14. Kumar R, Kohler E, Srivastava MB (2007) Harbor: software-based memory protection for sensor nodes. 2007 6th Int Symp Inf Process Sens Networks 340–349
15. NVD (2015) NVD. NIST
16. Shu R, Wang P, Gorski SA et al (2016) A study of security isolation techniques. ACM Comput Surv 49(50):1-50:37
17. Szekeres L, Payer M, Wei T, Song DX (2013) SoK: eternal war in memory. IEEE Symp Secur Priv 2013:48–62
18. Song Y (2017) On control flow hijacks of unsafe rust
19. Papp D, Ma Z, Buttyán L (2015) Embedded systems security: threats, vulnerabilities, and attack taxonomy. 2015 13th Annu Conf Privacy, Secur Trust 145–152
20. Larsen P, Homescu A, Brunthaler S, Franz M (2014) SoK: automated software diversity. IEEE Symp Secur Priv 2014:276–291
21. Tock (2015) TockOS. GitHub Repos
22. Clements AA, Almakhdhub NS, Bagchi S, Payer M (2018) ACES: automatic compartments for embedded systems. In: USENIX Security Symposium

23. Levy AA, Campbell B, Ghena B et al (2017) The case for writing a kernel in rust. APSys, In
24. Clements AA, Almakhdhub NS, Saab KS et al (2017) Protecting bare-metal embedded systems with privilege overlays. IEEE Symp Secur Priv 2017:289–303
25. LLVM (2015) Add support for embedded position-independent code (ROPI/RWPI). LLVM
26. Nilsson F, Adolfsson N (2017) A rust-based runtime for the internet of things
27. Hunt G, Letey G, Nightingale E (2017) The seven properties of highly secure devices