PISTIS: Trusted Computing Architecture for Low-end Embedded Systems

Michele Grisafi, *University of Trento;* Mahmoud Ammar, *Huawei Technologies;*
Marco Roveri and Bruno Crispo, *University of Trento*

https://www.usenix.org/conference/usenixsecurity22/presentation/grisafi

This paper is included in the Proceedings of the
31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the
31st USENIX Security Symposium is
sponsored by USENIX.

PISTIS: Trusted Computing Architecture for Low-end Embedded Systems

Michele Grisafi
University of Trento, Italy
michele.grisafi@unitn.it

Mahmoud Ammar
Huawei Technologies, Germany
mahmoud.ammar@huawei.com

Marco Roveri
University of Trento, Italy
marco.roveri@unitn.it

Bruno Crispo
University of Trento, Italy
bruno.crispo@unitn.it

Abstract

Recently, several hardware-assisted security architectures have been proposed to mitigate the ever-growing cyber-attacks on Internet-connected devices. However, such proposals are not compatible with a large portion of the already deployed resource-constrained embedded devices due to hardware limitations. To fill this gap, we propose PISTIS, a pure-software trusted computing architecture for bare-metal low-end embedded devices. PISTIS enables several security services, such as memory isolation, remote attestation and secure code update, while fully supporting critical features such as Direct Memory Access (DMA) and interrupts. PISTIS targets a wide range of embedded devices including those that lack any hardware protection mechanisms, while only requiring a few kilobytes of Flash memory to store its root of trust (RoT) software. The entire architecture of PISTIS is built from the ground up by leveraging memory protection-enabling techniques such as assembly-level code verification and selective software virtualisation. Most importantly, PISTIS achieves strong security guarantees supported by a formally verified design. We implement and evaluate PISTIS on MSP430 architecture, showing a reasonable overhead in terms of runtime, memory footprint, and power consumption.

1 Introduction

The last years have witnessed an increased use of embedded devices in many domains, ranging from tiny sensors to industrial control systems. These devices are increasingly connected to the Internet, realizing the idea of the Internet of Things (IoT) by connecting digital processes to the physical world. While this connectivity has enabled a vast array of value-added services in all applications, it has also opened the door to a wide variety of cyber-attacks. A high profile example is the Mirai Botnet [5], a clever malware that managed to turn vulnerable IoT devices, mostly specific types of smart cameras and DVRs, into zombies to mount the largest DDoS attack to date.

To detect potential attacks while protecting sensitive code and data, various hardware-assisted Trusted Execution Environments (TEEs)[1] have been proposed by both industry and academia, for high-end (e.g. Trusted Platform Modules (TPMs) [40], Intel SGX [15], and AMD SEV [22]), mid-range (e.g. ARM TrustZone [6]), and low-end platforms (e.g. ARM TrustZone-M [7], VRASED [30], SANCUS [29], and TyTan [10]). All of these TEEs provide confidentiality and integrity guarantees. The former is provided by enforcing strict access control on part of the memory through rich hardware features or some extensions to the original processor architecture, while the latter is realized by some sort of malware detection mechanisms, mainly remote attestation ($\mathcal{RA}$), that is supported on all architectures. $\mathcal{RA}$ is a security service that detects malware presence on a remote, potentially compromised, device, called prover, by verifying its software integrity using an integrity-ensuring function (e.g. a hash-based MAC) by a trusted party, called verifier.

Problem statement. Embedded devices are characterized according to many factors, among which, the support of hardware security features. In contrast to mid-range and high-end devices, low-end embedded devices are optimized for low cost, small size, and low power consumption. This makes them incapable of defending themselves against malware infestation attacks. In particular, a significant number of low-end embedded devices depend on Commercial Off-The-Shelf (COTS) Micro Controller Units (MCUs), which even lack basic hardware-based memory safety features such as memory protection units (MPUs), offering no (or, at best, little) security guarantees. Furthermore, the software in such devices runs bare-metal with no Operating System (OS) support. High profile examples include all AVR ATmega MCUs, many of the ARM Cortex-M family,[2] and all Flash-based MSP430 MCUs. Notable is that the underlying memory architecture of such MCUs is either the Von Neumann one [41], in which both

[1] Please note that we use the terms "Trusted Execution Environment" and "Trusted Computing Architecture" interchangeably.

[2] Please note that ARM Cortex-M0 has by default no MPU, whereas the MPU is optional in other ARMv7-based processors of the same family.

program code and data reside in a single address space, or the Harvard architecture [37], in which code and data memories are physically separated.

Initially, several software-based $\mathcal{R}$A protocols were proposed to detect malware presence on low-end devices [25, 33, 34]. Such protocols are hardware-independent. They rather rely on assumptions which, however, are hard to achieve in practice, such as silent adversary during $\mathcal{R}$A, optimal code, and one-hop communication [8]. This makes them vulnerable to several attacks [11], providing no certain security guarantees. More recently, various hardware-assisted $\mathcal{R}$A mechanisms with different assumptions and requirements have been proposed for embedded devices [10, 18, 23, 29, 30]. While such proposals provide strong security guarantees, their future availability in low-end devices is questionable due to several reasons. First, these proposals are not compatible with legacy MCUs as they require hardware modifications and thus replacing millions of the already deployed sensors and actuators. Second, changing vendor production lines as a result of hardware modification is not always easy and incurs extra costs. Third, pure hardware solutions, such as SANCUS [29], are inflexible in terms of patches in case of a vulnerability, requiring changing all affected platforms. As an example, a group of researchers exploited an unpatchable hardware design flaw in Xilinx 7-Series FPGA boards which led to a full break of the bitstream encryption, resulting in the total loss of authenticity and confidentiality [19]. Another example includes the removal of Intel Memory Protection Extensions (MPX) from all future Intel processors due to several design flaws [31].

Despite the limited initiatives to propose reliable software-based security services for low-end devices [2, 3, 24], existing proposals are limited to the Harvard architecture, which is simpler than the Von Neumann one. Furthermore, the security in such proposals is guaranteed under the assumption that the underlying embedded device lacks (or physically disables) Direct Memory Access (DMA), which is an important functionality that cannot be omitted in many low-end devices. Last, security services in such proposals execute atomically by disabling all interrupts, thus negatively influencing the correct functioning of safety-critical applications.

Contributions. This paper tackles the aforementioned issues by proposing PISTIS, [3] a pure-software trusted computing architecture for low-end Von Neumann-based embedded devices. PISTIS is built from the ground up, targeting embedded devices with limited or no hardware security features. It only requires a few kilobytes of the Flash memory of the corresponding device to hold its Trusted Computing Module (TCM). The TCM is a set of software functions that acts as a hypervisor, enabling different security services, such as memory protection, remote attestation, and secure code update. PISTIS provides strong security guarantees, comparable to those in hardware-assisted architectures, while securely supporting DMA and interrupts. Furthermore, PISTIS has the advantage of being portable to several MCU architectures because of its nature as a pure-software solution. In addition to the formally verified design, PISTIS is accompanied by an open-source implementation for the MSP430 family of MCUs [9].

In a nutshell, this paper makes the following contributions:
(1)* PISTIS*: To the best of our knowledge, this is the first design of a full-featured software-based trusted computing architecture that offers memory isolation, remote attestation, and secure code update services with strong security guarantees, targeting Von Neumann-based embedded devices.
(2) Formal Verification: We formally verify the correctness of the memory isolation design that provides the foundation for establishing the root of trust (RoT) in pure software.
(3) Open Source Release: We release PISTIS as an open-source C library along with a GCC compiler plugin (via [9]) with APIs that automate the deployment of PISTIS as well as the generation of PISTIS-compliant binary images.
(4) Extensive Evaluation: We evaluate PISTIS covering all relevant performance metrics, including runtime overhead, memory footprint, and power consumption, using a representative set of 13 applications.

Paper outline. The remainder of the paper is organized as follows. Section 2 reviews the related work. Preliminaries are presented in Section 3. Section 4 describes PISTIS in detail, whereas Section 5 highlights the methodology followed in verifying the design of the memory isolation feature employed by PISTIS. Implementation details and evaluation are reported in Section 6 and Section 7 respectively. Section 8 concludes and gives directions for future work.

2 Related Work

The constant threat of cyber-attacks on computing systems has driven the design of numerous security technologies, among which TEEs are one of the most widely proposed in the literature and most employed in diverse computing platforms. Existing TEEs generally depend on pure hardware or leverage a combination of software and hardware modules to protect one or multiple running applications by providing them with confidentiality and integrity guarantees.

In high-end (and some mid-range) platforms, various hardware-based TEEs have been designed and implemented for different processors, including Intel, ARM, and AMD. For instance, Intel Software Guard Extension (SGX) [15] extends the Instruction Set Architecture (ISA) of Intel processors to create hardware-enforced virtual containers, called *enclaves*, capable of protecting the integrity, confidentiality, and runtime state of internal applications without trusting any existing software module, e.g. the OS. ARM-TrustZone [6] is a well-known TEE for ARM processors, which provides a single hardware-based enclave, dividing the memory into two zones: secure and insecure. Thus, all sensitive apps run

[3] PISTIS was the personification of *good faith* in Greek mythology.

Table 1: PISTIS vs. State-of-the-art trusted computing architectures from various perspectives.

Architecture	HW modification	Memory Organization	DMA support	Interrupts support	Verified Design	Extensibility*
SANCUS [29]	Yes ✗	Von Neumann	No ✗	No ✗	No ✗	No ✗
VRASED [30]	Yes ✗	Von Neumann	Yes ✓	No ✗	Yes ✓	No ✗
SMART [18]	Yes ✗	Von Neumann	No ✗	No ✗	No ✗	No ✗
TrustLite [23]	Yes ✗	Harvard	No ✗	Yes ✓	No ✗	No ✗
TyTAN [10]	Yes ✗	Harvard	No ✗	Yes ✓	No ✗	No ✗
SμV [2]	No ✓	Harvard	No ✗	No ✗	No ✗	Yes ✓
PISTIS	No ✓	Von Neumann	Yes ✓	Yes ✓	Yes ✓	Yes ✓

*Extending or updating existing hardware-assisted architectures can only be considered for future devices

in a physically isolated secure memory area. Generally, existing solutions for high-end platforms depend on complex and expensive hardware features that cannot be supported on low-end devices due to cost and size constraints.

Given the lack of rich hardware features on low-end embedded platforms, various types of lightweight security architectures have been proposed. Initially, the main focus was on software-based solutions that enable security services such as remote attestation and secure code update without depending on hardware [25, 33–35]. In particular, such proposals depend on precise time measurements where the upper bound is estimated during the initialization of the embedded device (assuming time-space optimal code), or require filling the free memory space with true randomness in order to prevent malware from using it. As aforementioned, such solutions are vulnerable to some attacks as demonstrated in [11]. Furthermore, their security guarantees are uncertain due to depending on unrealistic assumptions, e.g. passive remote adversary, optimal code, one-hop communication, etc.

In contrast to software-based solutions, SANCUS [29] has been proposed as a lightweight security architecture for embedded devices that implements a pure-hardware trust anchor. Compared to other hardware-based TEEs, e.g. SGX [15], SANCUS is cheaper and more suitable for embedded devices. However, it still requires significant changes to the underlying architecture of these devices, increasing their cost. To overcome this limitation, several hybrid architectures have been introduced, depending on a hardware-software co-design [10, 18, 23, 30]. While there are several trade-offs between the designs of these hybrid approaches, they aim to provide the same security guarantees as hardware-based ones, while minimizing modifications to the underlying hardware. Although these approaches yield strong security guarantees and good performance on low-end IoT devices, they require customized hardware support on every device, which is either absent or too expensive to implement in certain scenarios. For instance, a wide spectrum of embedded devices that lack any hardware support is already employed in privacy-sensitive and safety-critical domains, making the adoption of such hybrid architectures impractical [27, 36]. Notable is that VRASED [30] is the only formally-verified hybrid architecture that provides a secure and sound remote attestation ($\mathcal{R}$A) service.

The security MicroVisor (SμV) [2] filled the sizeable gap between software-based and hardware-assisted security architectures by implementing a software-based memory isolation technique to isolate its RoT software (TCM) from the rest of other untrusted software modules running in the same address space. To do so, the SμV is inspired by the Software Fault Isolation (SFI) approach proposed by Wahbe et al. [42]. However, in contrast to all existing memory protection techniques that reuses SFI in high-end computing systems [38], the SμV does not trust the compiler toolchain, relying only on its TCM. Furthermore, it targets low-end Harvard-based MCUs. On top of memory isolation, the SμV provides various security services, such as remote attestation (e.g. SIMPLE [3]), secure erasure (e.g. SPEED [4]), and secure code recovery (e.g. VERIFY&REVIVE [1]). Nevertheless, the SμV does not support either DMAs or interrupts in the aforementioned security services. Notable is that the SμV's implementation (and not its design) has been formally verified w.r.t. to safety properties, targeting only the Harvard-based AVR architecture.

Although PISTIS shares similarities with SμV in terms of adapting and optimizing some of the SFI techniques in a new setting, it targets the Von Neumann architecture, which is more challenging than the Harvard one as clarified in Section 3.2. Furthermore, it aims for supporting DMA and interrupt operations with critical security services.

To sum up, Table 1 compares PISTIS with the state-of-the-art approaches.

3 Preliminaries

3.1 Scope of Embedded Devices

PISTIS targets tiny embedded devices that have small MCUs based on the Von Neumann architecture with little or no hardware security features. In general, such MCUs have a single core and feature Flash and SRAM memories. They execute instructions in place (in physical memory) and have no memory management unit (MMU) to support virtual memory. Some of them can support an MPU. However, our design and implementation neglect the existence of MPUs due to their shortcomings as clarified in [43]. In particular, PISTIS targets single-thread, yet multi-tasking bare-metal applications that are the most common ones in the IoT domain [14].

Our implementation is based on the MSP430 architecture. This choice is due to the wide use of this architecture in many IoT devices as well as research prototypes. Nevertheless, our design is applicable to other low-end MCUs in the same class, such as ARM Cortex-M.

3.2 Challenges in designing PISTIS

All MCUs used in small embedded devices (regardless of their vendors) are designed based on either of two memory architectures: Von Neumann or Harvard. In contrast to the SμV [2] that is proposed for the Harvard architecture, PISTIS targets the Von Neumann one. Both architectures are visualized in Figure 1. There are two main challenges in the Von Neumann architecture compared to the Harvard one. First, the Harvard architecture features a hardware isolation address space as both program (non-volatile Flash) and data (volatile SRAM) memories are physically separated, having different instructions for accessing them. This feature simplifies isolating part of the memory to host the trust anchor as the data memory is not executable, and thus no need to instrument its related instructions. This is not the case in the Von Neumann architecture where both program and data memories share the same memory address space and thus are executable, facilitating code injection attacks as basically any instruction could alter the state of the program memory, i.e. read from or write to it. This poses a challenge when designing PISTIS without incurring an intolerable performance overhead. Second, in contrast to the Harvard architecture, the Von Neumann one has a variable-length instruction set that would be exploited to break any software-based memory isolation technique by jumping into the middle of a multi-word instruction and executing one of its words as an unaligned instruction. This challenge cannot be handled by borrowing some of the well-known SFI techniques, i.e. as in the SμV [2], without considering a careful design to adapt them. To solve the first challenge while avoiding adding a high performance overhead, PISTIS focuses on checking the position of the Program Counter (PC) rather than instrumenting the entire instruction set. Also, PISTIS devises a special canary mechanism to mark legitimate jump targets, thus handling the second challenge. Furthermore, PISTIS, compared to all existing architectures regardless of their types, is the only solution that supports both DMA and interrupts while executing critical operations. Further details are provided in the next section.

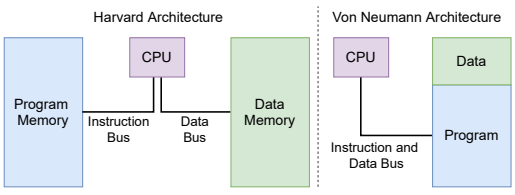


Figure 1: Harvard vs. Von Neumann architecture.

4 PISTIS

Overview. Systems security strongly relies on the concept of *trust* as lacking it renders any security service infeasible. Trust is typically assured via some sort of *memory isolation*, a mandatory security primitive for all security services. To this end, PISTIS follows the bottom-up approach to build a pure-software trusted computing architecture that is highly optimized for low-end embedded devices with a full support for DMA and interrupt operations. To achieve so, PISTIS initially deploys an initial code, called a Trusted Computing Module (TCM), that occupies part of the Flash memory including the bootloader area. The main goal of the TCM is to guarantee memory protection by isolating its memory area from other memory parts, creating two logically isolated memory zones: secure and insecure. PISTIS leverages the secure memory part to deploy two exemplar security services, namely remote attestation and secure code update. Please note that while PISTIS adapts some of the SFI techniques, i.e. binary re-writing, in its special domain, it does not depend on any hardware feature, such as segmentation registers in MPUs as in [20]. Furthermore, in contrast to standard SFI mechanisms [38], PISTIS represents a full-fledged trusted computing architecture, offering a run-time virtualization environment without trusting the compiler toolchain. The correctness of the memory isolation design in PISTIS is formally verified as clarified in Section 5.

In what follows, we outline our adversary model and then describe the design of PISTIS in detail.

4.1 Adversary Model

We consider a software-based adversary $\mathcal{A}$dv who has full access to the network or potentially presents inside the device itself in the form of malware. $\mathcal{A}$dv can eavesdrop on or tamper with traffic on any communication medium supported by the target device. However, she cannot launch Denial of Service (DoS) attacks. Protection against DoS attacks is out-of-scope as they do not tamper with the device memory. $\mathcal{A}$dv can also control any software deployed in the insecure memory part of the device. This includes trying to inject malicious code, reading or writing any memory address that is not explicitly protected, corrupting specific data, or manipulating I/O pins.

We assume that the TCM is initially and correctly installed on the embedded device by a trusted party. We also assume that the TCM is bug-free and does not contain any memory corruption vulnerabilities. [4] This means that $\mathcal{A}$dv cannot bypass any protection rules enforced by the TCM.

We rule out all physical and hardware-focused attacks. We consider that protection against physical attacks is an orthogonal problem and can be achieved, for instance, by deploying the device inside a tamper-proof protection shield [32].

[4] Likewise the work in [2], the memory-safety property can be achieved by formally verifying the code before deploying it.

4.2 PISTIS: From the Ground Up

In what follows, we will describe the main building blocks of PISTIS, namely the TCM and the accompanied compiler toolchain, that guarantee memory isolation, a key-enabling property for all trusted computing architectures. We then extend the described design by enriching the TCM with some security services, called Trusted Applications (TAs), i.e. remote attestation and secure code update, leveraging memory protection as a basis. All building blocks sum up PISTIS.

4.2.1 Memory Isolation: Design Rationale

The memory isolation property of PISTIS aims at enforcing memory protection to guarantee the integrity and confidentiality of PISTIS's TCM and its TAs. This is achieved by first deploying the TCM on the device using a physical programming device, e.g. JTAG, by a trusted party. The TCM is responsible for protecting itself against any untrusted software that is deployed on the same device afterward. To achieve so, the TCM requires any software to be deployed through it to verify its safety at the instruction level. The entire deployment will be rejected by the TCM if there is at least one unsafe instruction that violates the memory isolation property. PISTIS is accompanied by a PISTIS-enabled compiler toolchain that produces compatible binary images. However, this toolchain is untrusted as it is easy for $\mathcal{A}$dv to tamper with it. Therefore, the security of PISTIS depends on the load- and run-time verification that occurs on the device itself by the TCM.

Trusted Computing Module (TCM). The TCM is a set of software functions that reserves part of the non-volatile memory, including the bootloader area, to act as a hypervisor that fully manages access control to the entire memory area. In particular, the TCM consists of the following components:

- **Initial code**: a bootloader code that replaces the original one. When the MCU is powered on, this code decides whether to continue booting from the TCM memory or from the memory that holds untrusted software.
- **Loader/Verifier**: this software module is responsible for receiving the untrusted software image from the network interface and verifying whether it is PISTIS-compliant. If so, the software will be installed and activated on the device. Otherwise, it will be rejected and erased.
- **Virtualized Instructions**: a set of functions that represent safe equivalent versions to some potentially unsafe instructions that cannot be checked at load-time. During compilation, the PISTIS-enabled toolchain will replace each instruction with a hyper-call to a safe equivalent one that can be safely verified at runtime.
- **Helper modules**: other necessary helper functions such as the ones that read and write memory pages.

The TCM code runs in a privileged mode in the sense that it can perform any memory access operation. On the contrary, the untrusted software is subject to some restrictions that limit its access to some memory regions. Such restrictions are meant to enforce memory protection and are thus checked and enforced by the TCM. Figure 2 shows the memory layout of the MCU when activating PISTIS. The TCM is the first to be physically deployed in the PISTIS *Core* memory part. It then manages other deployments to maintain the shown layout. When deploying an application, the TCM verifies its instructions according to the following access policy (AP):

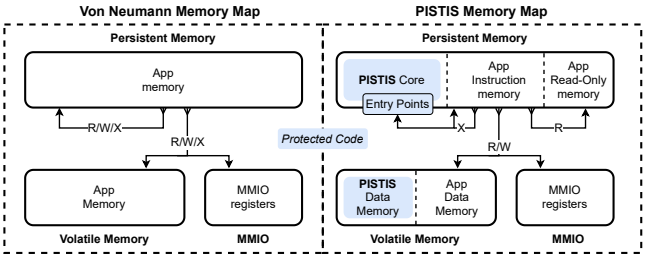


Figure 2: A standard memory map vs. the PISTIS-enabled memory map in a Von Neumann architecture. The latter also shows R/W/X privileges of the untrusted application.

- Read access is limited to Memory Mapped I/O (MMIO), Application Data and Application Read-Only memory.
- Write access is limited to MMIO memory [5] and Application Data memory.
- Jumps (to execute instructions) are limited to Application Instruction memory and specific entry points of the PISTIS Core memory.

In contrast to the Harvard architecture where maintaining memory isolation only requires taking care of control-transfer instructions, any instruction in the Von Neumann architecture can violate the aforementioned AP. Therefore, to adhere to such a policy, the TCM's Loader/Verifier should smartly verify each memory access (read/write) or control-transfer (jump/call) instruction of untrusted software before deploying it on the device. Initially, the TCM only knows the boundary of its non-volatile memory area and the dedicated space in the volatile one. To know the other boundaries shown in Figure 2, the PISTIS-enabled compiler toolchain produces some meta-data that is sent ahead of the deployed binary image. The TCM performs the verification process according to such meta-data. Please note that the values of such meta-data are accepted as long as they do not cross protected memory regions. With that in mind, the TCM installs the entire received binary image in the expected part of the non-volatile application memory. It then starts verifying it after disabling all interrupts to ensure atomicity. The following checks must be passed before the actual deployment of untrusted software:

- Instructions with a static addressing mode, whose target memory address is known, are checked and verified at load-time. These instructions can be either control-transfer or memory-access ones. They both must comply with the AP visualized in Figure 2 in the sense that:

[5] Exceptions can be made for critical MMIO registers.

  - control-transfer instructions can only target the memory area where the application will be installed or one of the entry points of the TCM. This check guarantees the Data Execution Prevention (DEP) property of the data memory since PC will not be allowed to jump there.
  - Write instructions can only target the permitted part of the volatile data memory (RAM) or (some) MMIO registers.
  - Read instructions can only target any write-permitted memory location or the application read-only memory.
- Instructions with a dynamic addressing mode whose target address is only known at runtime are replaced with static instructions in the form of hyper-calls to their secure virtualized versions (part of the TCM). If, at runtime, the target address is deemed to be unsafe, PISTIS performs a soft reset of the MCU to block this operation.

If the binary image contains at least one instruction that does not pass the above checks, it will not be deployed and accordingly erased. The list of instrumented instructions is described in Table 5 in Appendix A. Please note that PISTIS does not support self-modifying code in the sense that the untrusted software cannot directly write to its instruction memory area. However, if required, the untrusted software can invoke the TCM's Loader/Verifier module after installing the needed chunks of code in the non-executable data memory. The TCM's Loader/Verifier will relocate the installed chunks to the required memory location if they adhere to the aforementioned AP.

Modified Toolchain. PISTIS leverages a modified toolchain to transparently re-write each potentially unsafe dynamic instruction, and replace it with a safe virtualized equivalent that can be accessed via a call to a subroutine stored in the protected TCM memory area. The target address of the corresponding instruction is verified at runtime when the subroutine is invoked. The execution continues normally if it is valid. Otherwise, an MCU reset is triggered.

Figure 3 visualizes the modified compiler toolchain. We chose the open-source GCC compiler toolchain and modified it to suit our needs. Our modifications represent: (i) an instrumenter plugin that verifies the validity of static instructions and re-writes dynamic ones, and (ii) a custom linker script that maintains the required memory layout and resolves the addresses of valid TCM's entry points. The instrumenter module is placed between the compiler and assembler, targeting assembly instructions. During instrumentation, all instructions with a static addressing mode are left untouched as they are checked at load-time by the TCM on the device itself. All control-transfer and memory-access instructions with a dynamic addressing mode are re-written as previously clarified. Please note that $\mathcal{A}$dv cannot write hand-crafted assembly instructions or use her own toolchain without being detected

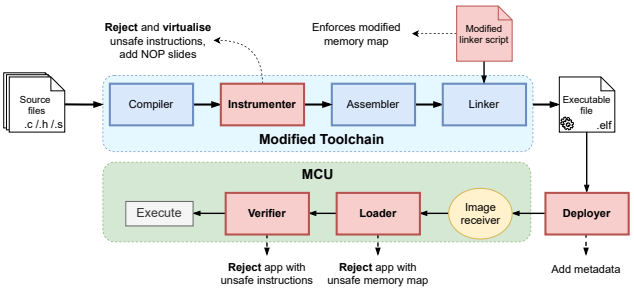


Figure 3: PISTIS-enabled Compiler Toolchain.

by the TCM. Application developers can use the PISTIS-enabled toolchain with the same ease of use as any other toolchain. No extra action is needed as all required steps are performed transparently. Furthermore, developers can simply allow their software to interact with the TCM by invoking any of its valid entry points.

Virtual Instructions. Virtual instructions are part of the TCM. They represent a safe replacement to some dynamic unsafe instructions, maintaining the same functionality and adhering to the aforementioned AP. This allows the TCM to verify the safety of the intended operation at runtime before executing the corresponding instruction.

Listing 1 shows an example of an unsafe dynamic CALL instruction that tries to jump to an address held by a register. Such an address is only known at runtime. When compiling with the PISTIS-enabled toolchain, such an instruction is replaced by a sequence of instructions shown in Listing 2. The main purpose of these instructions is to safely invoke an equivalent safe routine inside the TCM. This routine will check the validity of the target address before jumping to it. The instructions of such a routine are shown in Listing 3.

```
    ...
    CALL R10 //Dynamic call to an address in a register
```

Listing 1: An example of a dynamic unsafe CALL instruction.

```
    ...
    DINT // Disable interrupts to ensure atomicity
    MOV R10, R6 // copy target address to R6
    CALL #safe_call // Call to a TCM's safe virt. routine
    ...
```

Listing 2: A safe equivalent virtualization to the CALL instruction in Listing 1.

```
    ...
safe_call:
    CMP #topInstrMem, R6 // Check upper boundary
    JHS .stopExecution // MCU reset if AP is violated
    CMP #btmInstrMem, R6 // Check bottom boundary
    JL .stopExecution // MCU reset if AP is violated
    EINT //Enable interrupts after passing all checks.
    BR R6 // Jump to original destination
```

Listing 3: An example of a safe virtual function. `safe_call` checks the validity of the original destination before jumping.

4.2.2 Memory Isolation: Supported Operations

Direct Memory Access (DMA). DMA is a crucial feature that enables direct access to the memory without CPU intervention, yielding faster and efficient data transfer between the main memory and other external modules or I/O peripherals. However, DMA is rarely supported by TEEs since it allows bypassing the CPU-controlled access control mechanisms. Notable is that DMA operations in simple embedded systems are always preceded by a CPU-controlled initialization phase that entails writing some metadata, e.g. source and destination addresses, to some specific MMIO registers. Upon confirmation from the CPU, the DMA controller can execute independently to complete the entire operation. PISTIS is designed to support safe DMA operations by supervising the initialization phase. First, the instrumenter plugin in the PISTIS-enabled toolchain is configured to instrument the write instructions to the DMA-dedicated MMIO registers. Second, the Loader/Verifier module in the TCM is extended to check the existence of safe equivalents to such instructions by means of valid hyper-calls. Thus, safe DMA transfers can be guaranteed in a similar way to the control-transfer instructions, enabling execution if both the source and destination addresses are valid w.r.t. the aforementioned AP.

Interrupts. Interrupts enable the preemption of the normal application flow to execute user-defined functions, called Interrupt Service Routines (ISRs). These routines are fetched by the CPU depending on the Interrupt Vector Table (IVT), a priority-ordered list containing entry points to all defined ISRs. Supporting interrupt operations is important for many safety-critical application scenarios. Therefore, PISTIS supports preemptive execution of TCM's software modules except for virtual functions (as shown in Listing 3) as they are very small (each contains less than 10 instructions) and it is important for the security to execute them atomically.

To enable safe interruption, PISTIS supports a secure context switching mechanism through a chain of function-calls. To do so, PISTIS considers duplicating the IVT and moving both versions into its core memory (the TCM memory). All entry points inside the original IVT are overwritten to point to a unified secure ISR that is maintained inside the TCM memory as well. When invoked, this ISR backs up the MCU state to a fixed safe memory location. It also clears any sensitive data that is accessible by the untrusted software, e.g. contents inside registers. It then triggers an IVT lookup in the other non-modified version of the IVT to execute the target ISR by the interrupt operation. In principle, any ISR represents a set of instructions that is included as a part of the deployed binary image. Thus it can be instrumented (during compilation) and verified by the TCM at load-time. PISTIS instrumentation also adds a call to a virtual function (stored in the TCM memory) at the end of each ISR. When executed, this function safely restores the CPU state. Please note that the modified linker script in the PISTIS-enabled toolchain holds the new address of the modified IVT to maintain the intended functionality of compiled applications.

4.2.3 Memory Isolation: Variable Length Instructions

The Von Neumann architecture supports a variable-length instruction set, i.e. some instructions can be of variable lengths, holding one or more words. $\mathcal{A}$dv-s can maliciously leverage this feature to arbitrarily execute code and bypass the memory protection enforced by PISTIS. In other words, some benign-like instructions can behave maliciously by jumping into the middle of a multi-word instruction, located in the insecure memory zone. The target location can hold a value that could be executed as an instruction, allowing for illegal access to the protected memory area. Therefore, it is crucial for PISTIS to only allow jumping to verified instructions. While the TCM can easily check static jumps at load-time, a similar runtime check of dynamic jumps incurs a high runtime overhead.

To tackle this issue, we extend PISTIS's memory isolation design in two directions. First, we extend the functionality of the instrumenter plugin (in the PISTIS-enabled toolchain) to insert a special instruction, called *instruction canary*, before any valid address that can be a target for a potential jump. This instruction will be in the form of a NOP slide: a sequence of two NOP instructions. Given that our modification to the toolchain limits the number of dynamic jump instructions when possible, only a few NOP slides are inserted in each application. Second, the TCM is configured to take such NOP slides into account when virtualising jump instructions.When a dynamic jump executes at runtime, the corresponding virtual safe routine inside the TCM will check whether a NOP slide precedes the target address. If so, the jump is valid (i.e. the jump target is a permitted address inside the insecure memory area). Otherwise, an MCU reset is performed as a consequence of an invalid target address.

Considering an implementation for the MSP430 architecture, Figure 4 shows a snippet code, highlighting the working mechanism of the two-NOP canary. Please note that PISTIS does not enforce control flow integrity. Nevertheless, it guarantees that diverting control flow does not break the maintained memory isolation property.

4.2.4 Security Services

Leveraging memory isolation, PISTIS features two important security services, namely $\mathcal{R}$A and secure code update. Notable is that the design of other security services, e.g. secure erasure, can be easily supported in PISTIS.

Remote Attestation ($\mathcal{R}$A). Considering that the TCM memory is neither writable nor readable, we leverage it to design an $\mathcal{R}$A service. To do so, a secret key (that is pre-shared with the verifier) and an integrity verification function (based on MACs) are installed in the TCM memory. The first instruction of this function is considered a valid entry point to PISTIS. Whenever an attestation request is received, this function computes a digest (MAC value) of the entire memory

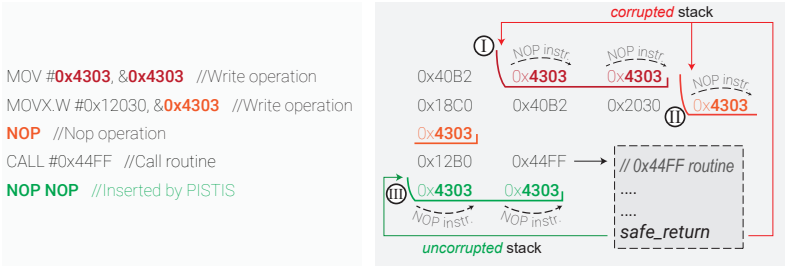


Figure 4: Binary code (right) of the assembly source code (left) has 3 NOP slides: a legitimate slide (III) after a `CALL` statement and two *accidental* slides (I) and (II) in the middle of two instructions. (III) is inserted by PISTIS. However, (I) and (II) are either maliciously inserted or accidentally derived from combinations of opcodes and operands. The `safe_return` virtual function only allows returning to a location with a slide. In case of a corrupted stack, the control flow can at most be diverted to another location containing an accidental slide. However, the instruction executed after the slide will always be safe since verified.

and then sends it back to the verifier for verification. Either a nonce or a time-stamp can be used to avoid replay attacks. Our $\mathcal{RA}$ is similar to SIMPLE [3]. However, it has the advantage of being interruptible as explained in Section 4.2.2.

Secure Code Update. We also extended the TCM to include a secure code update service that complies with SUIT [28], an IETF standard for software updates on IoT devices that requires maintaining authenticity, integrity, and confidentiality of the deployed software. By extending the cryptographic module of the TCM with a decryption function, our secure code update mechanism meets the above requirements. Considering a pre-shared secret key, the verifier has to send the software image encrypted along with a MAC value. The image will be deployed if it is decrypted and verified (in terms of matching MAC values and checking the compliance w.r.t. the AP of PISTIS) successfully.

5 Formalization

Formally verifying both the design correctness and implementation are important to provide strong security guarantees w.r.t. the aforementioned threat model (Section 4.1). Given that verifying the implementation is platform-dependent and requires continuous efforts with each software update to the PISTIS code, we leave it as future work. We rather focus on verifying the correctness of the design as it is platform-independent and gives more trustworthiness to the proposed solution.

In order to formalize the PISTIS design and memory map layout described in Figure 2, and prove it preserves the memory isolation, we need first to introduce some basic concepts. Let us assume for simplicity that we have a memory M that contains the code and where data can also be stored, and that there are a finite number of registers for the program counter `PC`, the stack pointer `SP`, and for intermediate results (e.g. $\texttt{R}_\texttt{j}$ for some `j`). Moreover, let us also assume there is an interrupt vector table (`IVT`) that contains the addresses of the first instruction of the interrupt service routines (ISRs).

Any application can be thought as composed of sequences and combinations of the following *primitive instructions*:

Definition 1 (Primitive (unsafe) instructions)**.** The set of *primitive (unsafe) instructions* is the following (we assume the address i is a valid address for the memory M):

- `Read(`M,i`)` that reads the content of memory M at address i: it denotes $M[i]$;
- `Write(`M,i,V`)` that writes V in memory M at address i: it denotes $M[i] = V$;
- `Goto(`M,i`)` that modifies the program counter `PC` to contain the address i of the given memory M: it denotes `PC =` i;
- `End` that terminates the execution of the entire application;
- Primitive operations (e.g. `and`, `add`, `comparison`) operating on registers/constants, and assignment (`=`) to a register.

An *application* P is a sequence and combination of the above primitive instructions, together with the `IVT` table specifying the addresses of the interrupt service routines.

We can safely assume that all the instructions of a typical MCU can be written in terms of these basic primitives. Let us consider for instance some instructions taken from the MSP430 MCU's family:

- `CALL funcname`: the execution of this instruction stores in the stack (in the memory) the current `PC`, modifying the `SP` to create space for storing the `PC`, and then modifies the `PC` to point to the address corresponding to `funcname` in the memory. Thus it corresponds to `SP = SP-1`, to create space in the stack, followed by `Write(M,SP,PC)`, to write the `PC` in the stack, followed by `Goto(M,funcname)` to update the `PC` and continue the execution from address `funcname`.
- `RET`: corresponds to `R = Read(M,SP)` followed by `SP = SP+1`, and finally `Goto(M,R)`, using the value stored in register `R`.

The primitive instructions also allow the modelling of more complex instructions like `PUSH` and `POP` in a very similar fashion, as well as different addressing modes (e.g. indexed, symbolic, indirect, absolute as for instance supported by the MSP430 MCU's family). For instance,

- `MOV 0x22(R3), 0x10(R4)` corresponds to `Write(M,0x10 + R4, Read(M, 0x22 + R3))`;

- `ADD 0x22(R3), 0x10(R4)` corresponds to `Write(M, 0x10 + R4, Read(M, 0x22 + R3)+ Read(M, 0x10 + R4))`;
- `MOV @R4, &0x3000` corresponds to `Write(M, 0x3000, Read(M, R4))`;
- `MOV 0x22, &0x3000` corresponds to `Write(M, 0x3000, Read(M, PC + 0x22))`.

Given the above basic concepts, to proceed in the formalization of the PISTIS design, the memory map layout described in Figure 2 (right), and the access policy AP (for reading, writing, and jumping), we need to: i) refine the memory M distinguishing between the persistent memory (M_P), the volatile memory (M_V), and the memory mapped IO (M_{MIO})[6]; ii) explicitly introduce the finite set $EnPo$ = { `i : word[N]` } of addresses for the PISTIS core instruction area as specified by the access policy; iii) and finally introduce the V_{IVT} : `array [K] of word[N]` - the interrupt vector `IVT` of size K. Moreover, we also need:

- utc_b, utc_e - start and end addresses of the PISTIS core;
- aim_b, aim_e - start and end addresses of the App Instruction Memory;
- $arom_b$, $arom_e$ - start and end addresses of the App Read-Only Memory;
- $utdm_b$, $utdm_e$ - start and end addresses of the PISTIS Data Memory;
- adm_b, adm_e - start and end addresses of the App Data memory;
- $mmio_b$, $mmio_e$ - start and end addresses of the MMIO Registers.

The following constraint formalizes the memory layout and the non-overlapping of the different memory areas.

$$0_N < utc_b < utc_e < aim_b < aim_e < arom_b < arom_e \leq 2_N^{N-1}$$
$$0_N \leq utdm_b < utdm_e < adm_b < adm_e \leq 2_N^{N-1}$$
$$0_N < mmio_b < mmio_e \leq 2_N^{N-1}$$

To model the constraint that the execution of PISTIS core instructions only happens through pre-defined Entry Points, we need a predicate $EP(i)$ which is true iff the address i is $utc_b \leq i \leq utc_e$ and i is an Entry Point address $EP_j \in EnPo$ for the PISTIS core memory.

$$EP(i) \leftrightarrow ((utc_b \leq i \leq utc_e) \wedge (\bigvee_{EP_j \in EnPo} i = EP_j))$$

Then, to formalize the read/write/jump policies as those enforced in the Figure 2 (right), we define the following terms:

$$AP_R(i,M) \leftrightarrow \begin{pmatrix} (M = M_P \rightarrow (arom_b \leq i \leq arom_e)) & \wedge \\ (M = M_V \rightarrow (adm_b \leq i \leq adm_e)) & \wedge \\ (M = M_{MIO} \rightarrow (mmio_b \leq i \leq mmio_e)) & \end{pmatrix}$$

$$AP_W(i,M) \leftrightarrow \begin{pmatrix} (M = M_V \rightarrow (adm_b \leq i \leq adm_e)) & \wedge \\ (M = M_{MIO} \rightarrow (mmio_b \leq i \leq mmio_e)) & \end{pmatrix}$$

$$AP_X(i,M) \leftrightarrow ((M = M_P) \wedge (EP(i) \vee (aim_b \leq i \leq aim_e)))$$

Moreover, we need also to ensure that each element of the `IVT` table is a valid address w.r.t. the access policy, so that:

$$AP_{IVT}(M) \leftrightarrow \forall i. AP_X(V_{IVT}[i], M)$$

We denote with AP the access policy resulting from the memory layout, from the read/write/jump constraints, and from the fact that each $i \in EnPo$ is such that $utc_b \leq i \leq utc_e$. Given the above formalizations, we can formally define when an application preserves memory isolation w.r.t. a given access policy AP.

Definition 2. Given a memory layout and an access policy AP, an application P *preserves memory isolation w.r.t. the access policiy* AP iff any of its read/write/jump instructions in all possible executions is such that the addresses used in such instructions satisfy the given access policy AP.

We remark that the primitive instructions in Def. 1 do not enforce particular restrictions on the addresses where to read-/write or jump (it suffices them being valid addresses). As thoroughly discussed, if the addresses of the application are constant, then checking whether the application preserves memory isolation is trivial. Indeed, it suffices to check whether each address in the application satisfies AP. However, as noted, in many cases such addresses are results of the execution of the application itself, thus the check can only be performed during the execution of the application.

To enforce memory isolation, for a given access policy AP, we can define safe variants of the read/write/jump instructions that will guarantee at runtime that no violation of the access policy occurs.

Definition 3 (Safe primitive instructions)**.** Given an access policy AP, the *safe* read/write/jump *primitive instructions w.r.t. AP* are:

- $\texttt{Read}_{\texttt{sf}}(M,i)$ that reads the content of memory M at address i if address i is such that read policy $AP_R(i,M)$ holds, otherwise it ends execution;
- $\texttt{Write}_{\texttt{sf}}(M,i,V)$ that writes V in memory M at address i if address i is such that write policy $AP_W(i,M)$ holds, otherwise it ends execution;
- $\texttt{Goto}_{\texttt{sf}}(M,i)$ that modifies the program counter `PC` to contain the address i if address i is such that branch access policy $AP_X(i,M)$ holds, otherwise it ends execution.

The following theorem holds for the *safe primitive instructions* defined as above.

Theorem 1. Given an access policy AP, the *safe primitive instructions* $\texttt{Read}_{\texttt{sf}}$, $\texttt{Write}_{\texttt{sf}}$, and $\texttt{Goto}_{\texttt{sf}}$ w.r.t. such AP preserve memory isolation and do not allow to violate AP.

[6] Without loss of generality, we consider each memory as an array of size $2^{\texttt{N}}$ of bit vectors of size `N` (i.e. `array word[N] of word[N]`), although only a small part might be used. The addresses are bit vectors of size `N` (i.e. `word[N]`. We use 0_N to represent the unsigned word of size N corresponding to value 0, similarly 2_N^{N-1} to represent the unsigned word of size N corresponding to value 2^{N-1}.

The proof directly follows from the definition of the *safe primitive instructions* w.r.t. an AP (see Appendix C.2 for the proof). If AP is violated, then each instruction results in ending the execution of the entire application, thus preventing access to memory areas forbidden by AP. On the other hand, if the address satisfies AP, instruction specific access to the specified memory location is allowed.

Given this, we can prove that any application P such that i) $AP_{IVT}(M)$ holds (i.e. each address $i \in V_{IVT}$ satisfies $AP_X(M,i)$), and ii) uses only the safe primitive instructions, preserves memory isolation.

Theorem 2. Let AP be an access policy, P be an application specified with the set of (unsafe) primitive instructions. Let P_{sf} be the application obtained from P by replacing each of the unsafe primitive instructions with the corresponding safe primitive instruction. If $AP_{IVT}(M)$ holds, then P_{sf} preserves memory isolation, preventing accessing memory addresses or executing code that violates AP.

The proof is by induction on the structure of the application P_{sf} leveraging on Theorem 1 (see Appendix C.1 for the proof). We remark that enforcing each element of the `IVT` to satisfy AP ensures that also the interrupt routines are located in memory locations allowed by AP. This requirement can be relaxed by not only rewriting the unsafe read/write/jump instructions with the corresponding safe ones, but also adding for each interrupt routine a new wrapping interrupt routine and modifying the `IVT` to point to the respective wrapping interrupt routine. Each wrapping interrupt routine first checks that the target address is a safe one, and if so, it does the jump to the original address in the non-modified `IVT` copy. Otherwise, it ends the execution.

6 Implementation

A prototype of PISTIS is implemented for the MSP430 architecture from Texas Instrument [17]. MSP430 MCUs are based on a Von Neumann memory model, and they are widely used in many critical application domains. For instance, they are employed in implantable medical devices (IMDs), e.g. pacemakers, that have support for standard interfaces for wireless communication [16, 21, 36]. They also have been a target for many research prototypes, including SANCUS [29], SMART [18], and VRASED [30].

In particular, we implemented PISTIS on top of the MSP430F5529 MCU, which features ∼132 kB of FLASH, ∼8 kB of SRAM, and up to an 8 MHz of CPU speed using internal oscillators. We used HMAC-SHA256 and ChaCha20 from the HACL library [44] to implement $\mathcal{R}$A and secure code update services respectively.

6.1 PISTIS in Numbers

Our PISTIS implementation is two-fold: we provide a software-based TCM for MSP430F5529 MCU and a GCC compiler plugin that produces PISTIS-compliant binary images. The TCM is modular in the sense that it is composed of a core and various TAs that can be deployed independently by adjusting some configurations in a custom Makefile which serves as an API. The TCM core, including a boot code, Loader/Verifier, virtual functions, the interrupt context switch, and some helper functions, is composed of 810 lines of C code along with 703 lines of Assembly. The cryptographic primitives used, namely HACL HMAC-SHA2 and ChaCha2, comprise 944 lines of code, extracted from the HACL library [44]. Leveraging these primitives, $\mathcal{R}$A implementation includes 78 lines of code, whereas the secure code update mechanism is composed of 220 lines of C code. The MSP430 architecture features 60 main instructions, of which, we had to instrument 24 instructions to maintain strong isolation guarantees (further details can be found in Appendix A). We provide an API, comprising a 102-line Makefile in addition to an extended MSP430F5529 linker script with 257 lines, that fully automates the deployment of the TCM on the target device. Furthermore, our GCC plugin serves as another API to fully automate the production of PISTIS-compliant binary images. It comprises a 71-line Makefile that transparently modifies 15 lines of the original linker script of applications and executes 745 lines of python scripts to re-write instructions and embed meta-data. Furthermore, the user API includes a 236-line python script that automates the deployment of produced binaries using serial communication.

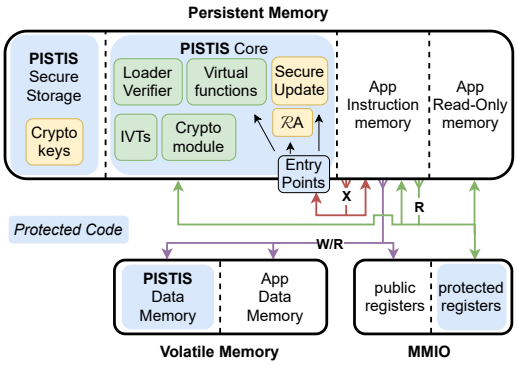


Figure 5: PISTIS memory map on MSP430 architecture.

6.2 Memory Map in Practice

Section 4 elaborated on the design rationale of enforcing memory isolation between software modules. As a consequence, Figure 2 visualized the resulted memory map w.r.t. the employed access policy (AP). Given that the AP is mainly concerned with read, write, and execute rights, the number of application instructions requiring virtualisation might vary depending on the application nature. To optimise PISTIS for our target MSP430 architecture, and thus reduce the number of virtualised instructions, we follow a slightly different memory map that, nevertheless, has equivalent security guarantees to the one shown in Figure 2. The new memory map, visualized in Figure 5, introduces a secure storage segment, where all the sensitive data, including cryptographic keys, are

stored. This memory part is only accessible by the PISTIS core, relaxing the application access policy as follows: (i) Read access is extended to cover the PISTIS Core, App Instruction memory, and PISTIS Data memory; and (ii) Write access is extended to cover PISTIS Data memory.

While our general design solely relies on the virtualisation and verification of the application code to ensure compliance with the AP of PISTIS, our implementation leverages two existing commodity hardware features in MSP430 MCUs: the Bootloader Section (BSL) and the Flash memory controller. Further details follow.

6.2.1 Secure Storage using BSL

The BSL is a small memory segment, located as a part of the Flash memory, whose confidentiality and integrity are hardware-enforced by the MCU. It triggers a reset at any illegal access. A legal access occurs through a few entry points, i.e. the Z-area, which can be configured during the physical deployment of PISTIS. Leveraging its intrinsic properties, we customise part of this segment to form the PISTIS secure storage, thus blocking any access by the untrusted application. The Loader/Verifier module in PISTIS is only required to check that the application instructions do not jump to the Z-area. Only the TCM can freely access the Z-area. This allows PISTIS to safely enable read access to the rest of the Flash memory without breaking the security.

6.2.2 Integrity of PISTIS Core

While the untrusted software is allowed to read any part of PISTIS core memory, it must not be allowed to have write access to it in order to preserve the integrity of the TCM. One possibility is to follow the guidelines of our design and virtualise all write instructions. However, the MSP430 architecture offers a hardware feature that allows achieving the same goal with better performance. The Flash memory controller regulates all write accesses over the Flash. It must be set up via custom memory-mapped registers that enable (unlock) or disable (lock) writing to the Flash according to their loaded value. In particular, writing to the Flash will only have an effect if there is an instruction that unlocks the Flash controller, i.e. by writing a specific pre-defined byte sequence (password) into one of the controller registers. Thus, PISTIS prevents the untrusted application from writing to any part of the Flash memory by checking that none of its instructions try to unlock the Flash memory controller. This reduces the number of instructions that have to be virtualised and fully preserves the integrity of PISTIS core. Please note that the volatile Data memory (SRAM) is not controlled by the Flash memory controller. Therefore, to reduce the performance overhead, PISTIS allows applications to read from or write to any part of it. PISTIS rather prevents the leakage of any sensitive data by clearing all shared memory areas before any context switch from a privileged to a non-privileged mode.

7 Experimental Evaluation

The variety of embedded applications along with the lack of suitable benchmarks make the evaluation of a generic trusted computing architecture such as PISTIS non-trivial, requiring a careful selection of a representative set of applications that would reflect the overhead incurred by PISTIS from various perspectives.

We chose a set of 13 applications, considering two main factors: (i) the compatibility with our target experimental platform (MSP430F5529LP), and (ii) making a good balance between the more CPU-intensive and the more IO-intensive tasks that are typical for an embedded device. The source and main functionalities of each application are described in Appendix B. Our evaluation metrics include: memory footprint, execution time, deployment time, and power consumption.

NOTE. Unless otherwise specified, applications are compiled using `-O3` optimization flag and 8 MHz as a CPU speed.

7.1 Memory Footprint

Increased Size of Instrumented Applications. Needless to say that instrumentation adds extra bytes to the size of each application due to inserting extra instructions, e.g. NOP slides, virtual calls, etc.

Considering a GCC toolchain with a `-O3` optimization flag (that further optimizes the execution time), Table 2 shows the size differences (in bytes) between binaries compiled using a standard GCC toolchain (Orig.), and a PISTIS-enabled GCC toolchain (Mod.) from two perspectives:

- The first two columns compare the sizes of the produced ELF (Executable and Linkable Format) files. ELF is a common standard file format for executable files, object code, and shared libraries, which tells the bootloader how to parse the received image, extract the real binary, and deploy it. Therefore, the size of ELF binary images is larger than native binaries since they include many extra meta-data. In the case of PISTIS, our toolchain customizes the relevant included meta-data, resulting in the reduction of the size of ELF binaries by an average of 77.6% as shown in Table 2. Please note that in embedded systems, applications are compiled statically as the entire binary image is considered self-contained with no need for dynamically-linked libraries. Therefore, all the libraries used in any application can be compiled using the PISTIS's compiler framework.
- The other two columns in Table 2 compare the real binary sizes of both standard and instrumented binary images when deployed. In other words, the recorded sizes represent the amount of consumed Flash memory in bytes. The increase of binary sizes when considering PISTIS averages at 41.17%.

PISTIS memory footprint. As shown in Figure 5, PISTIS persistently occupies part of the Flash memory. The size of the entire PISTIS software is 19.5 kB (of which 11 kB

Table 2: ELF Binary size and memory footprint comparison between original applications (Orig.) and PISTIS-enabled ones (Mod.).

App	ELF Binary		Memory Footprint	
	Orig.	Mod.	Orig.	Mod.
SerialMSP	3884 B	412 B (**-89.39%**)	302 B	356 B (**+17.88%**)
CopyDMA	5764 B	694 B (**-87.96%**)	444 B	628 B (**+41.44%**)
XorCypher	5940 B	532 B (**-91.04%**)	247 B	475 B (**+92.31%**)
Bitcount	5664 B	1602 B (**-71.72%**)	3684 B	5462 B (**+48.26%**)
SHA-256	9448 B	5518 B (**-41.60%**)	1376 B	1546 B (**+12.35%**)
ML-acc	16616 B	9512 B (**-42.75%**)	6174 B	9452 B (**+53.09%**)
PrimeFactor	33200 B	3650 B (**-89.01%**)	2192 B	3286 B (**+49.91%**)
32bitMath	6036 B	822 B (**-86.38%**)	522 B	766 B (**+46.74%**)
16bitSwitch	3940 B	182 B (**-95.38%**)	102 B	126 B (**+23.53%**)
8bitMatrix	4640 B	916 B (**-80.26%**)	844 B	860 B (**+1.90%**)
MatrixMul	4324 B	572 B (**-86.77%**)	500 B	516 B (**+3.20%**)
firFilter	24912 B	5486 B (**-77.98%**)	3312 B	5430 B (**+63.95%**)
dhrystone	7840 B	2468 B (**-68.52%**)	1335 B	2411 B (**+80.60%**)
Average		**-77.60%**		**+41.17%**

for the TCM core), amounting to 14.7 % of the available Flash memory (∼132 kB) on the target MCU.

7.2 Deployment time

In PISTIS, deploying any application occurs in two phases. First, the binary image is transferred as a stream of packets, starting with some meta-data, i.e. size, Cyclic Redundancy Check (CRC) value, etc. Second, the TCM's Loader/verifier module is executed to verify both the integrity of the binary image w.r.t. the received CRC value and the safety of its instructions w.r.t. the AP of PISTIS. If both checks are passed, the control is then given to the first instruction of this application, indicating a successful deployment.

Considering a serial UART communication medium with a speed of 9600 bps, Table 3 compares the deployment overhead incurred in each phase when deploying applications using either a standard compilation framework or the PISTIS one. Due to the customized ELF binary format used by PISTIS that further optimizes the size, the transfer time is faster, reducing the overhead by an average of no less than 75.4 %. On the contrary, PISTIS increases the verification time by an average of around 37.56 % due to the load-time verification mechanism employed. Nevertheless, the total deployment time when considering PISTIS is less than the time consumed in normal deployment procedures, averaging at −73.63 %.

7.2.1 Secure Deployment Overhead

So far, the description above did not consider the properties of the secure code update mechanism recommended by SUIT [28]. It only took into account the validation of a CRC32 value. Our secure deployment mechanism entails encrypting the target binary image at the transmitter side and decrypting it at the receiver side. This is in addition to including a MAC value that reflects the integrity status of the image during transmission. Comparing to the non-secure deployment mechanism in PISTIS, the secure code update slightly increases the transfer-time overhead due to including some extra bytes

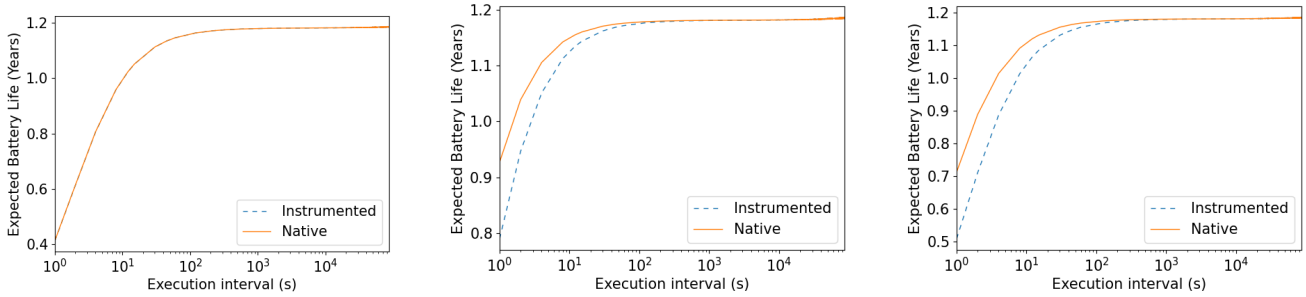
Table 3: Deployment (code update) time overhead when considering PISTIS (Mod.) compared to normal procedures (Orig.).

App	Transfer		Verification		Total
	Orig.	Mod.	Orig.	Mod.	
SerialMSP	3242 ms	347 ms	51 ms	62 ms	**-87.58 %**
CopyDMA	4809 ms	585 ms	92 ms	111 ms	**-85.80 %**
XorCypher	4957 ms	448 ms	42 ms	69 ms	**-89.66 %**
Bitcount	4727 ms	1340 ms	646 ms	913 ms	**-58.07 %**
SHA-256	7879 ms	4605 ms	212 ms	261 ms	**-39.86 %**
ML-acc	13849 ms	7930 ms	1534 ms	2109 ms	**-34.74 %**
PrimeFactor	27670 ms	3049 ms	597 ms	716 ms	**-86.68 %**
32bitMath	5035 ms	689 ms	113 ms	135 ms	**-83.99 %**
16bitSwitch	3289 ms	157 ms	29 ms	34 ms	**-94.24 %**
8bitMatrix	3871 ms	771 ms	172 ms	189 ms	**-76.26 %**
MatrixMul	3608 ms	482 ms	98 ms	196 ms	**-81.71 %**
firFilter	20767 ms	4577 ms	633 ms	910 ms	**-74.36 %**
dhrystone	6537 ms	2059 ms	210 ms	356 ms	**-64.21 %**
Average		**-77.52 %**		**+37.56 %**	**-73.63 %**

as a MAC value. In particular, 28 bytes are added to each binary image as a result of replacing CRC32 with an HMAC-SHA256 cryptographic primitive, increasing the overhead from −77.52 % to −77.09 %. Furthermore, the decryption and MAC verification increase the verification time overhead from 37.56 % to 78.82 %, resulting in an increase of the overall deployment time overhead from −73.63 % to −71.71 %. Nevertheless, the secure code update mechanism of PISTIS still outperforms the normal deployment procedure.

7.3 Execution time

In order to estimate the runtime overhead that PISTIS imposes on the normal execution of applications, we measured the difference between the time it takes to execute our test applications with and without PISTIS. Measurements are recorded at a CPU speed of 1 MHz, considering an average of 5 different test runs for each one. Table 4 shows that PISTIS increases the execution time by an average of 52.72 %. The maximum runtime overhead recorded is 127.32 %, whereas the minimum one is 0.17 %. The high heterogeneity of results is due to the nature of applications. For instance, memory- and CPU-intensive applications, e.g. CopyDMA and ML-acc, require virtualizing and verifying many instructions at runtime, incurring high runtime overhead. On the contrary, the I/O-intensive applications, e.g. SerialMSP, contain few unsafe instructions to virtualize, resulting in a slight increase of the execution time. It is important to mention that this is an acceptable price to pay for accommodating a full-featured trusted computing architecture without any hardware modification. We also note that using other optimization flags rather than `-O3` might optimize the execution time for some applications. For instance, using `-O2` optimization flag with the PrimeFactor application reduces the runtime overhead by 9 %, compared to the `-O3` version. Finally, we note that our $\mathcal{RA}$ consumes 7.4 seconds when computing MAC for a 64 kB memory block using the default clock speed (8 MHz).



(a) SerialMSP (b) SHA-256 (c) CopyDMA

Figure 6: An analysis of the power consumption of three different exemplar applications.

Table 4: Comparison between the execution time of original applications (Orig.) and the execution time of PISTIS-compliant ones (Mod.).

App	Normal Execution (Orig.)	PISTIS-enabled Execution (Mod.)
SerialMSP	334.1976 ms	335.325 ms (**+0.34%**)
CopyDMA	118.4960 ms	238.656 ms (**+101.40%**)
XorCypher	245.6500 ms	446.104 ms (**+81.60%**)
Bitcount	5.7520 ms	5.786 ms (**+0.59%**)
SHA-256	49.1888 ms	89.046 ms (**+81.03%**)
ML-acc	1456.9092 ms	3311.829 ms (**+127.32%**)
PrimeFactor	4.0810 ms	5.938 ms (**+45.50%**)
32bitMath	0.9310 ms	1.294 ms (**+38.99%**)
16bitSwitch	0.0050 ms	0.006 ms (**+20.00%**)
8bitMatrix	0.5760 ms	0.577 ms (**+0.17%**)
MatrixMul	0.3430 ms	0.344 ms (**+0.29%**)
firFilter	1093.5059 ms	2359.619 ms (**+115.78%**)
dhrystone	102.9200 ms	177.336 ms (**+72.30%**)
Average		**+52.72%**

7.4 Power consumption

IoT devices are often used in areas where a connection to a power line is simply unfeasible or expensive. Therefore, such devices depend on batteries as their main power source, requiring optimized energy usage to reduce maintenance costs.

Figure 6 compares the amount of consumed energy by applications compiled with and without PISTIS. Three different types of applications are considered: I/O-, memory-, and CPU-intensive. Considering a 3.0 V/3 A h battery, Figure 6 shows the impact on battery lifetime when considering different execution rates. In the worst-case scenario, our exemplar applications, namely SerialMSP, SHA-256 and CopyDMA, decrease the battery lifetime by $-0.22\,\%$, $-14.81\,\%$, and a $-28.69\,\%$ respectively, when running once a second. The battery lifetime degradation becomes lower than $0.01\,\%$ when running the aforementioned applications at rate of once every 2 min, 42 min, and 120 min, respectively. We also note that the battery would last for no less than 1 year when performing our $\mathcal{R}$A mechanism once an hour, considering any of the sample applications executing once every 2 minutes.

7.5 PISTIS in Practice

The reported runtime overhead might seem not acceptable in some scenarios. However, we note that this overhead does not reflect the complete image in real-world scenarios. To do so, we emulated a real case, in which we deployed PISTIS in an MSP430-based device equipped with a gyroscope sensor and a low-power IEEE 802.15.4 compliant radio. The main task of the device was to obtain 10 different measurements from the gyroscope (x,y,z coordinates), encrypt them, transmit the encrypted data over the network to a host controller, and wait for an acknowledgement. By measuring the time required to perform this entire set of actions (that constitutes the job of the IoT device), we noticed that out of a total of 276 ms as a complete execution time, only 6.02 % of the time was occupied by the software that should be instrumented by PISTIS. The majority of the time (93.98 %) was consumed by the network layer (that is hosted in another MCU and interfaced with through peripherals). This means that if PISTIS incurs 100 % performance overhead on top of the normal execution time of the corresponding software module, the overall performance overhead on top of the total execution time of the entire job will be not more than 9.5 %. Therefore, we believe that the runtime overhead incurred by PISTIS is reasonable in practice.

8 Conclusion & Future Work

This paper introduced PISTIS, a pure-software trusted computing architecture for low-end embedded devices. PISTIS provides memory isolation and other security services such as remote attestation and secure code update. It provides strong security guarantees while supporting interrupts and DMA operations. The correctness of the memory isolation design is formally verified. As future work, we want to formally verify the rest of the design of PISTIS architecture as well as its implementation. We further plan to design and implement a control flow attestation scheme on top of PISTIS.

Acknowledgments: The authors thank Prof. Thomas Mayor for shepherding this work. They also thank the anonymous reviewers for the valuable feedback. This research was partly supported by the EU-EIT Digital Project MCU Fortifier 2020-21395.

References

[1] Mahmoud Ammar and Bruno Crispo. Verify&Revive: Secure Detection and Recovery of Compromised Low-end Embedded Devices. In *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC)*, 2020.

[2] Mahmoud Ammar, Bruno Crispo, Bart Jacobs, Danny Hughes, and Wilfried Daniels. SμV—The Security MicroVisor: A Formally-verified Software-based Security Architecture for the Internet of Things. *IEEE Transactions on Dependable and Secure Computing*, 16(5):885–901, 2019.

[3] Mahmoud Ammar, Bruno Crispo, and Gene Tsudik. SIMPLE: A Remote Attestation Approach for Resource-constrained IoT devices. In *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*, pages 247–258. IEEE, 2020.

[4] Mahmoud Ammar, Wilfried Daniels, Bruno Crispo, and Danny Hughes. SPEED: Secure Provable Erasure for Class-1 IoT Devices. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 111–118, 2018.

[5] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the Mirai botnet. In *26th USENIX security symposium*, pages 1093–1110, 2017.

[6] ARM. TrustZone technology for Armv8-A. https://developer.arm.com/ip-products/security-ip/trustzone/trustzone-for-cortex-a#armv8-a, 2011. [Online; accessed 10-November-2020].

[7] ARM. TrustZone technology for Armv8-M. https://developer.arm.com/ip-products/security-ip/trustzone/trustzone-for-cortex-m, 2015. [Online; accessed 10-November-2020].

[8] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. A security Framework for the Analysis and Design of Software Attestation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1–12, 2013.

[9] Authors. PISTIS Source Code . https://github.com/CybersecurityUnitn/PISTIS.

[10] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. TyTAN: Tiny Trust Anchor for Tiny Devices. In *Proceedings of the 52nd Annual Design Automation Conference*, page 6, New York, New York, USA, jun 2015. ACM.

[11] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the Difficulty of Software-based Attestation of Embedded Devices. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 400–409, 2009.

[12] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014.

[13] Alessandro Cimatti, Raffaele Corvino, Armando Lazzaro, Iman Narasamdya, Tiziana Rizzo, Marco Roveri, Angela Sanseviero, and Andrei Tchaltsev. Formal Verification and Validation of ERTMS Industrial Railway Train Spacing System. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2012.

[14] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting Bare-metal Embedded Systems with Privilege Overlays. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 289–303. IEEE, 2017.

[15] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[16] Dave Muoio. DA warns of Bluetooth Low Energy vulnerability affecting connected medical devices. https://www.mobihealthnews.com/news/fda-warns-bluetooth-low-energy-vulnerability-affecting-connected-medical-devices, 2020. [Online; accessed 10-May-2021].

[17] John H Davies. *MSP430 microcontroller basics*. Elsevier, 2008.

[18] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *19th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2012.

[19] Maik Ender, Amir Moradi, and Christof Paar. The Unpatchable Silicon: A Full Break of the Bitstream En-

cryption of Xilinx 7-Series FPGAs. In *29th USENIX Security Symposium*, pages 1803–1819, 2020.

[20] Taylor Hardin, Ryan Scott, Patrick Proctor, Josiah Hester, Jacob Sorber, and David Kotz. Application Memory Isolation on Ultra-low-power MCUs. In *USENIX Annual Technical Conference (ATC)*, pages 127–132, 2018.

[21] imec. The medical implants of the future: faster, smarter and more connected. https://www.imec-int.com/en/imec-magazine/imec-magazine-april-2020/the-medical-implants-of-the-future-faster-smarter-and-more-connected, 2020. [Online; accessed 10-May-2021].

[22] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. *White paper*, 2016.

[23] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. TrustLite: A Security Architecture for Tiny Embedded Devices. In *Proceedings of the 9th European Conference on Computer Systems*, page 14, New York, NY, USA, 2014. ACM.

[24] Ram Kumar, Eddie Kohler, and Mani Srivastava. Harbor: Software-based Memory Protection for Sensor Nodes. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 340–349, 2007.

[25] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. SBAP: Software-Based Attestation for Peripherals. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, pages 16–29, Berlin, Heidelberg, 2010. Springer-Verlag.

[26] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - specification*. Springer, 1992.

[27] Francesca Meneghello, Matteo Calore, Daniel Zucchetto, Michele Polese, and Andrea Zanella. IoT: Internet of threats? A survey of practical security vulnerabilities in real IoT devices. *IEEE Internet of Things Journal*, 6(5):8182–8201, 2019.

[28] Brendan Moran, Milosch Meriac, Hannes Tschofenig, and David Brown. A Firmware Update Architecture for Internet of Things Devices. *Internet Engineering Task Force, Internet-Draft*, 2019.

[29] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. SANCUS: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium*, pages 479–498, 2013.

[30] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED: A verified hardware/software co-design for remote attestation. In *28th USENIX Security Symposium*, pages 1429–1446, 2019.

[31] Phoronix. Intel MPX Support Is Dead With Linux 5.6. https://www.phoronix.com/scan.php?page=news_item&px=Intel-MPX-Is-Dead, 2020. [Online; accessed 10-May-2021].

[32] Srivaths Ravi, Anand Raghunathan, and Srimat Chakradhar. Tamper Resistance Mechanisms for Secure Embedded Systems. In *17th International Conference on VLSI Design. Proceedings.*, pages 605–611. IEEE, 2004.

[33] A Seshadri, A Perrig, L van Doorn, and P Khosla. SWATT: Software-based Attestation for Embedded Devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 272–282. IEEE, may 2004.

[34] Arvind Seshadri, Mark Luk, and Adrian Perrig. SAKE: Software Attestation for Key Establishment in Sensor Networks. In *Distributed Computing in Sensor Systems*, pages 372–385, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[35] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SCUBA: Secure code update by attestation in sensor networks. In *Proceedings of the 5th ACM workshop on Wireless security*, pages 85–94, 2006.

[36] Muhammad Ali Siddiqi, Angeliki-Agathi Tsintzira, Georgios Digkas, Miltiadis G Siavvas, and Christos Strydis. Adding Security to Implantable Medical Devices: Can We Afford It? In *EWSN*, pages 67–78, 2021.

[37] T SPERRY. 386 VS 030-THE CROWDED FAST LANE. *DR DOBBS JOURNAL*, 13(1):16, 1988.

[38] Gang Tan et al. *Principles and Implementation Techniques of Software-based Fault Isolation*. Now Publishers, 2017.

[39] Texas Instrument. MSP430 Competitive Benchmarking. Application Report SLAA205B, 2006.

[40] Trusted Computing Group. TPM Main Specification Level 2 Version 1.2. http://www.trustedcomputinggroup.org/tpm-main-specification/, 2011. [Online; accessed 13-February-2017].

[41] John Von Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[42] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, dec 1993.

[43] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Good motive but bad design: Why ARM MPU has become an outcast in embedded systems. *arXiv preprint arXiv:1908.03638*, 2019.

[44] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806, 2017.

A List of Instrumented Instructions

The list of instrumented instructions is illustrated in Table 5.

B Applications descriptions

Our application test suite is composed of 13 different applications described in Table 6. We categorised the apps based on the type of operations performed. Specifically, we distinguish between I/O-oriented applications (i.e. using peripherals and performing memory operations), computational applications (i.e. performing CPU-intensive operations), and a mix of the two. Our application selection covers multiple domains, comprising two cryptographic primitives, several mathematics functions, a machine learning algorithm, a synthetic benchmark and applications in other domains. On top of 3 custom-made applications, appositely designed to stress different subsystems of our reference MCU, we selected 4 open-source applications from public repositories and 6 benchmark applications from the TI MSP430 Competitive Benchmark [39]. The latter is a representative selection[7] of the applications proposed by TI, comprising both simple and complex mathematical operations, and a synthetic benchmark.

C Formal Proofs of Theorem 1 and 2

C.1 Proofs for Theorem 1

In order to prove Theorem 1, we formalized the three operations in NUXMV [12] (a state of the art symbolic model checker), and for each of the three formalizations we considered some Linear Temporal Logic (LTL) [26] properties aiming at proving the correctness of the operations. In this approach we codify the three different operations as a sequential program encoded in NUXMV in form of Single Static Assignment [13] (as it is typically done in compilers and in software model checking), specifying for each value of the program counter: `s0` before the implicit condition checking the respective access policy i.e., $AP_R(i,M)$, $AP_W(i,M)$ or $AP_X(i,M)$); `s1` if the access point condition holds; `s2` right after the real operation on the memory for reading, writing, or modifying the program counter if correct; `end` representing the location to jump to if the access policy is violated. The variable `i` is a free variable that models the address we aim at reading from, writing to, and jumping to respectively. Then we have the three memories `PM` (Primary Memory), `VM` (Volatile Memory), and `MMIO`. `v` is the value to write in the memory in the case of the $\texttt{Write}_{\texttt{sf}}$.

Listing 5 contains the NUXMV code for the $\texttt{Read}_{\texttt{sf}}$. Here we model the $\texttt{Read}_{\texttt{sf}}$ with the `DEFINE ReadSV` that is a word of size `N+1` where the `N+1` bit is set to `0` if the access policy $AP_R(i,M)$ holds, and to `1` otherwise. The model is then complemented with three LTL properties. The first states that if the $AP_R(i,M)$ is always true, then the `state` can never take value `end`. The second property states that if the $AP_R(i,N)$ is always true, then the `N+1` is always `0` if the `state` is different from `s0` (i.e., the state before a $\texttt{Read}_{\texttt{sf}}$). Finally, the last LTL property states that if it is possible to reach a state where the violation of the $AP_R(i,N)$ holds, then it is possible to reach the state `end` (the state where the read has violated the access policy). In this model, the `i` variable can range over any possible value (there is thus an implicit universal quantification ($\forall$)).

Listing 4 reports the output of running NUXMV (taken from https://nuxmv.fbk.eu/) on this file. The execution has been done on a Linux laptop. NUXMV was able to prove (or disprove) the three LTL properties in few seconds.

```
computer_shell > nuXmv -int -dcx pistis_read.smv
*** This is nuXmv 2.0.0 (compiled on Oct 14 2019)
....
nuXmv > go_msat
nuXmv > check_ltlspec_ic3 -i
....
-- LTL specification ( G APr -> G state != end) is true
-- LTL specification ( G APr -> G (state != s0 -> ReadSV[32 : 32] = 0ud1_0)) is
      true
-- LTL specification ( F !APr -> F state = end) is false
-- (trace generation was suppressed)
nuXmv > quit
```

Listing 4: Run of NUXMV on the `pistis_read.smv` file to prove the correctness of the $\texttt{Read}_{\texttt{sf}}$ instruction.

These results show that the first two properties hold, while the last one is violated (as expected) to indicate that the only possible way to reach the `end` state is to violate the $AP_R(i,M)$ in a $\texttt{Read}_{\texttt{sf}}$. In this case, NUXMV can generate a trace showing how to reach that state (in the run for the sake of presentation we disabled the extraction of the counterexample, option `-dcx` at command line).

Similar considerations hold for the other two instructions.

[7] The original TI Competitive Benchmark contains several redundant applications. We chose the most relevant subset containing approximately one application per type.

Table 5: List of main MSP430 instructions with a brief description and whether they had been instrumented by PISTIS.

Instruction	Description	Instrumented	Why
ADC, ADCX	Add carry bit to destination	No ✗	Does not affect PISTIS
ADD, ADDX, ADDC, ADDCX	Add word to destination (w, w/o carry bit)	No ✗	Does not affect PISTIS
AND, ANDX	AND between source and destination	Yes ✓	Can unlock Flash controller
BIC, BICX, BIS, BISX, BIT, BITX	Clear, set, test bits of destination	No ✗	Does not affect PISTIS
BR	Jump to destination word	No ✗	Does not affect PISTIS
CALL	Call a function	Yes ✓	Can break application boundaries
CLR, CLRX, CLRC, CLRN, CLRZ	Clear destination or carry, negative, zero bits	No ✗	Does not affect PISTIS
CMP, CMPX	Compare source and destination	No ✗	Does not affect PISTIS
DADC, DADCX, DADD, DADDX	Add word to destination (w, w/o carry bit)	No ✗	Does not affect PISTIS
DEC, DECX, DECD, DECDX	Decrement, double decrement destination	No ✗	Does not affect PISTIS
DINT, EINT	Disable, enable interrupts	No ✗	Does not affect PISTIS
INC, INCX, INCD, INCDX	Increment, double increment destination	No ✗	Does not affect PISTIS
INV, INVX	Invert destination	No ✗	Does not affect PISTIS
JC, JHS, JEQ, JZ, JQE, JL, JMP, JN, JNC, JLO, JNZ, JNE	Jump to destination with a condition	Yes ✓	Can break application boundaries
MOV, MOVX	Copy source to destination	Yes ✓	Can break application boundaries, Can unlock Flash controller
NOP	No operation	No ✗	Does not affect PISTIS
POP, POPX, POPM	Pop value, values, from stack to register, registers	Yes ✓	Can break application boundaries, Can unlock Flash controller
PUSH, PUSHX, PUSHM	Push register, registers, to the stack	Yes ✓	Can unlock Flash controller
RET, RETI	Fetch return address from the stack	Yes ✓	Can break application boundaries
RLA, RLAM, RLAX, RLC, RLCX, RRA, RRAM, RRAX, RRC, RRCX, RRUM, RRUX	Rotate destination	No ✗	Does not affect PISTIS
SBC, SBCX	Subtract borrow bit from destination	No ✗	Does not affect PISTIS
SETC, SETN, SETZ	Set carry, negative, zero bits on status register	No ✗	Does not affect PISTIS
SUB, SUBX, SUBC, SUBCX	Subtract source to destination (w, w/o carry bit)	Yes ✓	Can unlock Flash controller
SWPB, SWPBX	Swap bytes of destination	No ✗	Does not affect PISTIS
SXT, SXTX	Extend sign of destination	No ✗	Does not affect PISTIS
TST, TSTX	Test destination	No ✗	Does not affect PISTIS
XOR, XORX	XOR between source and destination	Yes ✓	Can unlock Flash controller

Table 6: An overview of the main functionalities of the applications used in evaluating PISTIS.

App	Category	Domain	Source	Description
SerialMSP	I/O	Inter-device communication	Custom made	Set up an UART connection with a terminal and initiate the download of chunks of data to be stored on the RAM.
CopyDMA	I/O	Data migration / backup	Custom made	Initiate the migration of chunks of data from FLASH to RAM employing both CPU and DMA. The transferred data is checked for errors to verify its integrity.
XorCypher	Computational	Crypto	Custom made	Encodes various data streams with a custom-XOR stream cipher, saving the new encoded data to protect its confidentiality.
Bitcount	Computational	Benchmark	OpenSource https://github.com/embecosm/mibench	Perform different bit-based intensive computations over data arrays to test the computational power of the CPU.
SHA-256	mix	Crypto	OpenSource https://github.com/amosnier/sha-2	Compute the hash of various strings using the SHA-256 algorithm, storing it alongside the strings for integrity protection.
ML-acc	mix	Machine Learning	OpenSource https://github.com/CMUAbstract/dino	Train an activity recognition machine learning model to classify new accelerometer measurements into "shaking" and "still" activity.
PrimeFactor	mix	Mathematics	OpenSource https://github.com/TheAlgorithms/C	Calculate the prime factorisation for various input numbers to test the computational power of the CPU.
TI-32bitMath	Computational	Mathematics	TI Benchmark	Perform several 32-bit math operations on various input data.
TI-16bitSwitch	Computational	Operational	TI Benchmark	Test various switch cases using different 16-bit data.
TI-8bitMatrix	I/O	Operational	TI Benchmark	Backup the data from 8-bit based matrixes in RAM.
TI-MatrixMul	Computational	Mathematics	TI Benchmark	Perform different multiplications between matrixes in RAM, saving the new data.
TI-firFilter	mix	Signal processing	TI Benchmark	Benchmark an FIR filter of order 17 with various input data contained in arrays of 16-bit values.
TI-dhrystone	mix	Benchmark	TI Benchmark	Dhrystone synthetic benchmark

Listing 6 is the NUXMV code for proving correctness of the $\texttt{Goto}_{sf}$, while the one for the $\texttt{Write}_{sf}$ instruction can be found available on the public repository of PISTIS [9].

C.2 Proof of Theorem 2

Base case: There are four base cases each constituted respectively by: i) the NOP no-op instruction that does not perform any access to the memory neither for writing nor reading nor executing; ii) the $\texttt{Read}_{sf}$; iii) the $\texttt{Write}_{sf}$; iv) the $\texttt{Goto}_{sf}$. The last three instructions preserve memory isolation as proved in Theorem 1. The NOP preserves memory isolation since it does not access memory neither for writing nor for reading nor for executing. Thus the single instruction program preserves memory isolation.

Step case: Let us assume a program P preserves memory isolation. Let $P' = P$; `inst` be a program obtained by adding an instruction `inst` immediately after the last instruction of P. The possible instructions `inst` are: `NOP`, $\texttt{Read}_{\texttt{sf}}$, $\texttt{Write}_{\texttt{sf}}$, and $\texttt{Goto}_{\texttt{sf}}$. These instructions preserve memory isolation (given the base case and Theorem 1), thus given that P preserves memory isolation and the fact that the possible extension of the program P (i.e., the program P') also preserves the memory isolation, we can conclude that the theorem holds.

```
-- To run it:
-- shell > nuXmv -int
        pistis_read.smv
--
-- At the nuXmv prompt issue
        following commands:
-- go_msat; check_ltlspec_ic3 -i;
        quit
--
MODULE main
  VAR PM : array word[32] of
        word[32]; -- Persistent
        memory
  VAR VM : array word[32] of
        word[32]; -- Volatile
        memory
  VAR MMIO : array word[32] of
        word[32]; -- MMIO
  VAR mem_k : {_PM, _MMIO, _VM};
        -- kind of memory
  VAR state : {s0, s1, s2, end}; --
        Possible values of the PC
  VAR PC : word[32]; -- The
        program counter;
  VAR v : word[32]; -- Value to
        write
  VAR i : word[32]; -- Address to
        read from/write to

  -- Memory layout as mandated
        by the policy
  VAR EPadd : array word[3] of
        word[32]; -- List of entry
        points
  DEFINE utc_b := 0h32_00000010;
  DEFINE utc_e := 0h32_00000100;
  DEFINE aim_b := 0h32_00001000;
  DEFINE aim_e := 0h32_00010000;
  DEFINE arom_b := 0h32_00100000;
  DEFINE arom_e := 0h32_01000000;
  DEFINE utdm_b := 0h32_00000010;
  DEFINE utdm_e := 0h32_00000100;
  DEFINE adm_b := 0h32_00001000;
  DEFINE adm_e := 0h32_00010000;
  DEFINE mmio_b := 0h32_00000010;
  DEFINE mmio_e := 0h32_00000100;

  INVAR
    0h32_00000000 < utc_b & utc_b <
          utc_e & utc_e < aim_b &
    aim_b < aim_e & aim_e < arom_b &
          arom_b < arom_e &
    arom_e < 0h32_FFFFFFFF;
  INVAR
    0h32_00000000 < utdm_b & utdm_b
          < utdm_e &
    utdm_e < adm_b & adm_b < adm_e
          & adm_e < 0h32_FFFFFFFF;
  INVAR
    0h32_00000000 < mmio_b & mmio_b
          < mmio_e & mmio_e <
          0h32_FFFFFFFF;

  -- Access policy
  DEFINE APr :=
    (((mem_k = _PM ) -> ((arom_b <=
          i) & (i <= arom_e))) &
    ((mem_k = _VM ) -> ((adm_b <=
          i) & (i <= adm_e ))) &
    ((mem_k = _MMIO) -> ((mmio_b
          <= i) & (i <= mmio_e))));

  DEFINE APw :=
    (((mem_k = _VM ) -> ((adm_b <=
          i) & (i <= adm_e))) &
    ((mem_k = _MMIO) -> ((mmio_b
          <= i) & (i <= mmio_e))));
```

```
DEFINE APx :=
  ((mem_k = _PM ) & (EP | ((aim_b
        <= i) & (i <= aim_e))));

DEFINE EP := (((utc_b <= i) & (i
      <= utc_e)) &
  ((i = READ(EPadd, 0d3_0)) | (i =
        READ(EPadd, 0d3_1)) |
   (i = READ(EPadd, 0d3_2)) | (i =
        READ(EPadd, 0d3_3)) |
   (i = READ(EPadd, 0d3_4)) | (i =
        READ(EPadd, 0d3_5)) |
   (i = READ(EPadd, 0d3_6)) | (i =
        READ(EPadd, 0d3_7))));

-- Read_sf(M, i) := if
      (APr(i,M)) return M[i]
      else goto end;

ASSIGN
  init(state) := s0;
  next(state) := case
                   state = s0 & APr :
                         s1;
                   state = s1 & APr :
                         s2;
                   state = s2 : s2;
                   TRUE : end;
                 esac;

  DEFINE ReadSV := case
    state = s0 & APr : 0d33_0;
    state = s1 & APr : case
                   mem_k = _PM : 0d1_0
                         :: READ(PM, i);
                   mem_k = _VM : 0d1_0
                         :: READ(VM, i);
                   mem_k = _MMIO :
                         0d1_0 ::
                         READ(MMIO,
                         i);
                   TRUE :
                         0h33_FFFFFFFF;
                 esac;
    state = s2 : 0d1_0 ::
          0h32_FFFFFFFF;
    state = end : 0d33_0;
    TRUE : 0d33_0;
  esac;

  -- If the APr is always true, then
        there is not a possibility to
  -- reach the end state.
  LTLSPEC
    G(APr) -> G(state != end)

  -- If the APr is always true, and
        we are in any state
        different from
  -- s0, then the error flag bit is
        always 0
  LTLSPEC
    G(APr) -> G(state != s0 ->
          ReadSV[32:32] = 0d1_0)

  -- If APr is violated, then the
        state eventually become
        end,
  -- i.e. end is only reachable if
        APr is violated
  LTLSPEC
    F(!APr) -> F (state = end)
```

Listing 5: The encoding to prove the correctness of the $\texttt{Read}_{\texttt{sf}}$

```
-- To run it:
-- shell > nuXmv -int
        pistis_goto.smv
--
-- At the nuXmv prompt issue
        following commands:
-- go_msat; check_ltlspec_ic3 -i;
        quit
--
MODULE main
  VAR PM : array word[32] of
        word[32]; -- Persistent
        memory
  VAR VM : array word[32] of
        word[32]; -- Volatile
        memory
  VAR MMIO : array word[32] of
        word[32]; -- MMIO
  VAR mem_k : {_PM, _MMIO, _VM};
        -- kind of memory
  VAR state : {s0, s1, s2, end}; --
        Possible values of the PC
  VAR PC : word[32]; -- The
        program counter;

  VAR v : word[32]; -- Value to
        write

  VAR i : word[32]; -- Address to
        read from/write to

  -- Memory layout as mandated
        by the policy
  VAR EPadd : array word[3] of
        word[32]; -- List of entry
        points
  DEFINE utc_b := 0h32_00000010;
  DEFINE utc_e := 0h32_00000100;
  DEFINE aim_b := 0h32_00001000;
  DEFINE aim_e := 0h32_00010000;
  DEFINE arom_b := 0h32_00100000;
  DEFINE arom_e := 0h32_01000000;
  DEFINE utdm_b := 0h32_00000010;
  DEFINE utdm_e := 0h32_00000100;
  DEFINE adm_b := 0h32_00001000;
  DEFINE adm_e := 0h32_00010000;
  DEFINE mmio_b := 0h32_00000010;
  DEFINE mmio_e := 0h32_00000100;
  DEFINE END := 0h32_10000000;

  INVAR
    0h32_00000000 < utc_b & utc_b <
          utc_e & utc_e < aim_b &
    aim_b < aim_e & aim_e < arom_b &
          arom_b < arom_e &
    arom_e < 0h32_FFFFFFFF;
  INVAR
    0h32_00000000 < utdm_b & utdm_b
          < utdm_e &
    utdm_e < adm_b & adm_b < adm_e
          & adm_e < 0h32_FFFFFFFF;
  INVAR
    0h32_00000000 < mmio_b & mmio_b
          < mmio_e & mmio_e <
          0h32_FFFFFFFF;

  -- Access policy
  DEFINE APr :=
    (((mem_k = _PM ) -> ((arom_b
          <= i) & (i <= arom_e))) &
    ((mem_k = _VM ) -> ((adm_b <=
          i) & (i <= adm_e ))) &
    ((mem_k = _MMIO) -> ((mmio_b
          <= i) & (i <= mmio_e))));

  DEFINE APw :=
    (((mem_k = _VM ) -> ((adm_b <=
          i) & (i <= adm_e))) &
    ((mem_k = _MMIO) -> ((mmio_b
          <= i) & (i <= mmio_e))));

  DEFINE APx :=
    ((mem_k = _PM ) & (EP | ((aim_b
          <= i) & (i <= aim_e))));

  DEFINE EP := (((utc_b <= i) & (i
        <= utc_e)) &
    ((i = READ(EPadd, 0d3_0)) | (i =
          READ(EPadd, 0d3_1)) |
     (i = READ(EPadd, 0d3_2)) | (i =
          READ(EPadd, 0d3_3)) |
     (i = READ(EPadd, 0d3_4)) | (i =
          READ(EPadd, 0d3_5)) |
     (i = READ(EPadd, 0d3_6)) | (i =
          READ(EPadd, 0d3_7))));

  -- goto_sf(M, i, v) := if
        (APx(i,M)) PC = i else
        goto end;

  ASSIGN
    init(state) := s0;
    next(state) := case
                     state = s0 & APx :
                           s1;
                     state = s1 & APx :
                           s2;
                     state = s2 : s2;
                     TRUE : end;
                   esac;
  INIT
    PC != END;
  ASSIGN
    next(PC) := case
                  state = s0 & APx : PC;
                  state = s1 & APx : i;
                  state = s2 : PC;
                  TRUE : END;
                esac;

  -- If the APx is always true,
        then there is not a
        possibility to reach the
        end state.
  LTLSPEC
    G(APx) -> G(state != end)

  -- If the APx is always true,
        then the PC never
        assumes value END
  LTLSPEC
    G(APx) -> G (state != s0 -> PC
          != END)

  -- If APx is violated, then the
        state eventually become
        end,
  -- i.e. end is only reachable if
        APx is violated
  LTLSPEC
    F(!APx) -> F (state = end)
```

Listing 6: The encoding to prove the correctness of the $\texttt{Goto}_{\texttt{sf}}$