Untrusted Code Compartmentalization for Bare Metal Embedded Devices

Liam Tyler and Ivan De Oliveira Nunes

***Abstract*—Micro-controller units (MCUs) implement the de facto interface between the physical and digital worlds. As a consequence, they appear in a variety of sensing/actuation applications from smart personal spaces to complex industrial control systems and safety-critical medical equipment. While many of these devices perform safety- and time-critical tasks, they often lack support for security features compatible with their importance to overall system functions. This lack of architectural support leaves them vulnerable to run-time attacks that can remotely alter their intended behavior, with potentially catastrophic consequences. In particular, we note that, MCU software often includes untrusted third-party libraries (some of them closed-source) that are blindly used within MCU programs, without proper isolation from the rest of the system. In turn, a single vulnerability (or intentional backdoor) in one such third-party software can often compromise the entire MCU software state. In this article, we tackle this problem by proposing, demonstrating security, and formally verifying the implementation of *UCCA*: an Untrusted Code Compartment Architecture. *UCCA* provides flexible hardware-enforced isolation of untrusted code sections (e.g., third-party software modules) in resource-constrained and time-critical MCUs. To demonstrate *UCCA*'s practicality, we implement an open-source version of the design on a real resource-constrained MCU: the well-known TI MSP430. Our evaluation shows that *UCCA* incurs little overhead and is affordable even to lowest-end MCUs, requiring significantly less overhead and assumptions than the prior related work.**

***Index Terms*—Compartmentalization, embedded systems, hardware security, memory protection.**

I. Introduction

EMBEDDED systems have become critical components of many applications, including cyber–physical systems (CPSs) and the Internet of Things (IoT). Normally, these devices feature one or more resource-constrained Micro-Controller Units (MCUs) responsible for interfacing with the physical world (i.e., sensing and actuation). MCUs are often designed to minimize cost, size, and energy consumption. As such, they usually run software in place (physically from program memory) and lack virtual memory and other forms of isolation commonly found in higher-end devices.

Manuscript received 5 August 2024; accepted 10 August 2024. This work was supported in part by the NSF under Award SaTC-2245531. This article was presented at the International Conference on Embedded Software (EMSOFT) 2024 and appeared as part of the ESWEEK-TCAD special issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: Liam Tyler.)*

The authors are with the Computing and Information Sciences Ph.D. Program, Rochester Institute of Technology, Rochester, NY 14623 USA (e-mail: lgt2621@rit.edu; ivanoliv@mail.rit.edu).

Digital Object Identifier 10.1109/TCAD.2024.3444691

Due to their budgetary limitations, MCUs are often left vulnerable to run-time exploits [1], [2], [3], [4], [5] (for instance, triggered by buffer overflow vulnerabilities [6], [7]). Run-time attacks allow an adversary to remotely alter the intended behavior of a program during its execution. Without proper isolation, a single run-time vulnerability could give an adversary full control over the device [8], [9]. This can be used to spoof sensor data, bypass safety checks, ignore remote commands, and ignore scheduled task deadlines. For instance, a compromised patient-monitoring system implemented using MCUs could fail to alert medical personnel in case of an emergency [10] or cause a denial of service [11]. Similarly, vulnerable industrial control sensors could be used to run machines at unsafe speeds and damage equipment (e.g., as in the Stuxnet attack [12]).

Some MCUs (e.g., in the ARM Cortex-M family [13]) support rudimentary isolation to mitigate run-time attacks. Privilege levels [14], [15] allow the MCU to run software as either privileged or unprivileged. The MCU also restricts how the privileged code can be called. This enables the isolation of privileged code from unprivileged code. Thus, unprivileged run-time vulnerabilities cannot access the privileged functionality. While useful, this mitigation is limited as unprivileged vulnerabilities can still compromise all unprivileged code. Similarly, privileged vulnerabilities can reach all privileged and unprivileged software. Thus, vulnerabilities within a privileged function still result in a full system compromise.

Memory Protection Units (MPUs) allow for isolation between the privileged and unprivileged layers and further restrictions within each layer, by enforcing read, write, and execute permissions to a fixed number of memory segments. This allows more restricted compartments within the unprivileged layer, however, MPUs are configurable by the privileged software. As such, they cannot restrict privileged code, as any compromised privileged code could misconfigure the MPU. To make matters worse, the privileged layer must implement several low-level system functions, including all Interrupt Service Routines (ISRs) and respective drivers [15], [16], Direct Memory Access (DMA) management [17], real-time task scheduling [18], and more. This contributes to a large and complex Trusted Computing Base (TCB) that often relies on multiple untrusted third-party software modules and libraries. MPU-based protection often also requires disabling interrupts for unprivileged software creating a conflict between real-time requirements and security for the MCU software.

Motivated by this pressing issue, we propose, design, implement, and formally verify *UCCA*: an Untrusted Code

1937-4151 © 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Compartment Architecture. *UCCA* is a lightweight hardware-based memory isolation method that enables the definition of arbitrary-sized memory segments for untrusted code (e.g., third-party software) at device loading time (i.e., whenever physically programmed via USB, J-TAG, etc.). At run-time, *UCCA* monitors CPU signals to actively prevent malicious behavior within the untrusted sections from escalating to the remainder of the MCU.

Unlike current bare-metal approaches (e.g., MPUs) that isolate trusted functionality from the rest of the software, *UCCA* instead isolates untrusted code. Since, run-time attacks typically originate from well-known code sections (e.g., I/O functions or third-party libraries), untrusted code sections can be identified predeployment. Through isolation, *UCCA* limits the reach of exploits to their own context. Attempts to obtain similar guarantees with existing hardware lead to large memory and run-time overheads, limits its applicability to unprivileged code only, and may require disabling interrupts preventing asynchronous event handling (see Section III-C for details). In contrast, *UCCA* can isolate untrusted privileged code (such as drivers) and does not require disabling interrupts to enforce isolation. *UCCA* also enables finer-grained isolation that can be used jointly with existing hardware to further isolate unprivileged applications from their own untrusted code sections and third-party libraries. *UCCA* is designed as a hardware monitor that runs independently and in parallel with the MCU core. Therefore, no software (including privileged code) can misconfigure *UCCA*'s protections at run-time. Furthermore, *UCCA* incurs little execution time overhead (for marshaling data into isolated compartments) and maintains support for interrupts. In sum, this article's anticipated contributions are threefold:

1) Proposal and design of a lightweight hardware-based architecture for isolation of untrusted code sections in resource-constrained MCUs. This prevents the escalation of run-time vulnerabilities to the entire system. *UCCA* includes support for the isolation of the untrusted interrupts and untrusted privileged code sections.
2) Implementation and formal verification of *UCCA* atop an open-source version [19] of the well-known TI MSP430 MCU. *UCCA*'s prototype is publicly available at [20].
3) Evaluation of *UCCA* prototype and comparison to related approaches [21], [22], [23] in terms of hardware overhead. Along with *UCCA*'s open-source release, we implement sample attack programs, that show how their escalation is detected and prevented by *UCCA*.

II. Background

A. Scope of MCUs

This work focuses on resource-constrained embedded MCUs. These are single-core devices, executing instructions physically from program memory (i.e., at "bare metal"), and lacking a Memory Management Unit (MMU) to support virtual memory. We target these devices because an architecture that is simple and cost-effective enough for the lowest-cost MCUs is adaptable for higher-end devices with higher hardware budgets (whereas the reverse is often more challenging). In addition, the relative simplicity of these devices enables us to reason about them formally and verify *UCCA* security properties. With these premises in mind, we implement our *UCCA* prototype atop the TI MSP430; a well-known low-end MCU. This choice is also motivated by the availability of an open-source version MSP430 hardware from OpenCores [19]. Nevertheless, *UCCA*'s design and assumptions are generic and should also apply to other MCUs.

B. Linear Temporal Logic and Formal Verification

Computer-aided formal verification typically involves three steps. First, the system of interest (e.g., hardware, software, and protocol) is described using a formal model, e.g., a Finite State Machine (FSM). Second, properties that the model should satisfy are formally specified. Third, the system model is checked against these formally specified properties. This can be done via Theorem Proving [24] or Model Checking [25]. We use the latter to verify *UCCA*'s implementation.

We formally specify desired *UCCA* properties using Linear Temporal Logic (LTL) and implement *UCCA* hardware as FSMs using the Hardware Description Language (HDL) Verilog [26]. Hence, *UCCA*'s hardware FSM is represented by a triple: (σ, σ_0, T), where σ is the finite set of states, $\sigma_0 \subseteq \sigma$ is the set of possible initial states, and $T \subseteq \sigma \times \sigma$ is the transition relation set, which describes the set of states that can be reached in a single step from each state.

To verify the implemented hardware against the LTL specifications we use the popular model checker NuSMV [27]. For digital hardware described at Register Transfer Level (RTL) (the case in this work) conversion from HDL to NuSMV models is simple. Furthermore, it can be automated [28] as the standard RTL design already relies on describing hardware as FSMs. LTL specifications are useful for verifying sequential systems. In addition to propositional connectives, conjunction ($\land$), disjunction ($\lor$), negation ($\neg$), and implication ($\rightarrow$), LTL extends propositional logic with *temporal quantifiers*, thus enabling sequential reasoning. Along with the standard future quantifiers, *UCCA*'s verification also uses Past-Time LTL [27], [29] to reason about past system states. Specifically, *UCCA* formal specifications and respective verification rely on the following LTL temporal quantifiers:

1) $\mathbf{X}\phi$ – neXt ϕ: holds if ϕ is true at the next system state.
2) $\mathbf{G}\phi$ – Globally ϕ: holds if for all future states ϕ is true.
3) $\psi \mathbf{W} \phi$ – ψ Weak Until ϕ: holds if ψ is true for *at least* all states until ϕ becomes true or ψ is globally true if ϕ never becomes true.
4) $\mathbf{Y}\phi$ – Yesterday ϕ (a.k.a. previous ϕ): holds if ϕ was true in the previous system state.

C. Run-Time Exploits and Software Isolation

Run-time software attacks allow an adversary ($\mathcal{A}\mathsf{dv}$) to remotely alter the intended behavior of a program. The majority of program instructions execute sequentially, however so-called *branching instructions* (e.g.,: function calls, returns, if statements, and loops) can alter this sequence. Thus, branching instructions define the program's intended *control flow*. If certain vulnerabilities are present, $\mathcal{A}\mathsf{dv}$ can

hijack these instructions and change the software's intended behavior. For example, buffer overflows [6], [7] overrun a buffer's allocated memory to corrupt adjacent stack memory and potentially the current function's return address. As such, $\mathcal{A}dv$ can craft malicious oversized buffer inputs, overwrite return addresses, and force a jump to some $\mathcal{A}dv$-defined address. Consequently, this leads to well-known attacks, such as control flow hijacking [30], [31], code injection [32], [33], and Return Oriented Programming (ROP) [2], [3], [4], [5]. For an overview of run-time software vulnerabilities and their consequences see [1].

The recurrence of run-time exploits has led to various mitigation (see Section VII). Among them, isolation techniques are the predominant method to prevent programs from interfering with each other. In particular, they aim to protect a given process from tampering by another malicious/compromised task executing on the same device. Higher-end devices (e.g., general-purpose computers and servers) rely on virtual memory to enforce interprocess isolation. On these devices, unprivileged processes (typically all processes but the Operating System (OS)) can only stipulate memory accesses via virtual addressing. An MMU in the CPU translates each virtual access to a physical address in real-time. These translations are only configurable by privileged software (typically the OS). Therefore, as long as the MMU is securely configured, unprivileged processes cannot interfere with each others' control flow, code, or data. Notably, MMU-based isolation assumes the OS is vulnerability-free. This implies a large TCB, often including low-level code (i.e., device drivers), and has led to numerous attacks on OS implementations [34], [35], [36].

Regardless of their benefits or shortcomings, the hardware cost of virtual memory and MMU-based isolation is prohibitive for MCUs. Lower-end MCUs often have no support for isolation (e.g., TI MSP430 and AVR ATMega) whereas higher-end MCUs (e.g., some ARM Cortex-M MCUs) feature less expensive MPUs. MPUs are hardware monitors that configure physical memory regions with different read, write, and execute permissions for privileged and unprivileged software. MPUs can protect security-critical code against tampering by enforcing 1) read-only permissions for critical code sections and 2) data execution prevention for data segments. Similar to MMUs, MPUs are configured by privileged software (e.g., an embedded OS, such as FreeRTOS [18]). Thus, MPUs must also trust the OS, as the OS can freely configure the MPU.

Some higher-end MCUs are also equipped with TrustZone-M [37]. TrustZone is an architectural extension that divides MCU hardware, software, and data into a Secure and Nonsecure world. The Secure world is an isolated execution environment for security-critical software. The Secure world can only be called from the Nonsecure world through secure entry points called Nonsecure Callables (NSCs). To enable this separation, TrustZone adds new hardware extensions to the MCU. The Secure Attribution Unit (SAU) and Implementation Defined Attribution Unit (IDAU) [38] mark memory as Secure, Nonsecure, and Nonsecure Callable. This assigns the memory to the corresponding world and marks it as an NSC, respectively. The IDAU defines a base memory configuration that the SAU can overwrite to elevate their definitions. While the SAU and IDAU divide memory between two worlds, they do not provide further separation within each world nor prevent vulnerabilities in the Secure world from compromising the Nonsecure world. As such, the SAU and IDAU enforce configurations defined by an additional level of privilege.

We note that, the premise of the existing controls is that security-critical sections can be determined a priori. *UCCA (this work) is rooted in the different and complementary premise that untrusted code segments, i.e., those more likely to contain software vulnerabilities can also be enumerated a priori*. We stress that this does not require that *UCCA* pinpoints/identifies vulnerabilities themselves (a much harder task) but rather allows for defining "less trusted" code sections. As discussed earlier, run-time attacks typically originate from well-known code sections, e.g., low-level I/O manipulation exposed to malformed/malicious inputs and third-party (often closed-source) code. Thus, these components are good candidates for compartmentalization in *UCCA*. Once untrusted code segments are defined, *UCCA* prevents attacks in these regions (if any) from escalating to the remainder of the system. Therefore, *UCCA* can work in tandem with existing hardware to not only protect security-critical code from the rest of the MCU software but also ensure that likely vulnerable code segments, if/when exploited, cannot escalate to the rest of the system. Importantly, *UCCA*'s design allows isolation within privileged software for increased protection even against privileged vulnerabilities.

III. **UCCA** Overview

UCCA is a hardware monitor that isolates untrusted code compartments (*UCC*s) from the rest of the system. What constitutes untrusted code varies with application domains and developer-defined security policies. As such, *UCC*s are flexible to allow for different isolation cases. *UCC*s contain executables and are defined by their first and last addresses in physical memory; namely $UCC_{\min}$ and $UCC_{\max}$ (recall from Section II-A that MCUs execute instructions in-place, physically from program memory). *UCC* locations in memory are configurable and can have arbitrary size. All *UCC* definitions (($UCC_{\min}$, $UCC_{\max}$) pairs) are stored in a reserved and protected part of physical memory denoted the "Configuration Region" (*CR*). Their values are loaded to *CR* when the MCU is physically programmed/flashed and *UCCA* prevents *CR* from being overwritten at run-time. Thus, once defined, *UCC*s cannot be changed or disabled by any software.

To isolate each *UCC*, *UCCA* monitors CPU signals to enforce two properties, Return and Stack Integrity. Return integrity prevents invalid returns (as well as any other malicious jumps) from *UCC*s. Whenever execution enters a *UCC*, *UCCA* saves a copy of the return address. Then, when *UCC* finishes running, *UCCA* enforces that execution returns to this previously saved value. This prevents any control flow attacks within *UCC* from escalating to the rest of the system. Stack integrity creates an isolated stack frame for each *UCC*. This isolated frame allows code within *UCC* to write to the stack and heap while preventing modifications to stack memory belonging to functions external to *UCC*. Stack integrity also ensures the stack pointer is properly set when returning from

UCC. This stops attempts to corrupt data in use by other functions in the same device.

Despite these restrictions, *UCC*s remain interruptable. If a *UCC* is interrupted, *UCCA* loosens return integrity to allow execution to jump to the associated ISR. Once outside the *UCC*, stack integrity is disabled allowing the interrupt to edit the stack as needed. While interrupted, *UCCA* maintains the saved return address and isolated stack frame. Then, when execution returns to *UCC*, return and stack integrity are re-enforced. A malicious interrupt could abuse this behavior to break *UCCA*'s protections. Nonetheless, *UCCA* allows any untrusted ISR to also be confined within a dedicated *UCC* thus preventing control flow and stack tampering that could otherwise originate from the malicious ISR. While isolated, these ISRs remain interruptable allowing for nested interrupts.

If either of the aforementioned rules are violated, *UCCA* triggers an exception, preventing *UCC* execution from continuing. Since, our prototype MCU, the MSP430, treats all exceptions with a device reset, we use the same mechanism. However, other types of (software-defined) exception handling are also possible. While resetting the device can impact availability, any $\mathcal{A}$dv can already use run-time attacks to force device resets (e.g., by jumping to an invalid address among other exceptions). Thus, *UCCA*'s exception handling does not provide $\mathcal{A}$dv with more capabilities than already available.

A. Adversary ($\mathcal{A}$dv) Model

We assume an $\mathcal{A}$dv that attempts to fully compromise the MCU software state. We assume that one or more *UCC* resident programs contain vulnerabilities that enable control flow hijacks, ROP, and code injection attacks. Code external to *UCC*s is assumed to be benign. We emphasize that being privileged does not imply being trusted. Thus, risky privileged code can be defined as untrusted in *UCCA*. $\mathcal{A}$dv's goal is to exploit *UCC*-resident and vulnerable code to compromise (otherwise benign) code outside *UCC*s, by tampering with its control flow, program memory, or data. In other words, $\mathcal{A}$dv aims to escalate a *UCC*-resident vulnerability to compromise the rest of the system. Physical/hardware tampering attacks are out of the scope of this article. In particular, we assume that $\mathcal{A}$dv cannot modify/disable the physical hardware, induce hardware faults, or bypass *UCCA* formally verified hardware-enforced rules. Protection against physical $\mathcal{A}$dv and hardware-invasive attacks is considered orthogonal and can be obtained via physical access control and standard tamper-resistance techniques [39].

B. *UCCA* Architecture

Fig. 1 depicts *UCCA*'s architecture. *UCCA* adds a new hardware monitor, denoted HW-Mod to the underlying MCU. *UCCA* also reserves a dedicated region in memory to store *UCC* configurations, i.e., *CR*. *CR* stores the address of each region's first and last instruction. The size of *CR* varies with the number of simultaneous *UCC*s supported. HW-Mod monitors the values within *CR* to create isolated regions in memory. To detect violations, HW-Mod also monitors six additional signals from the MCU's core:

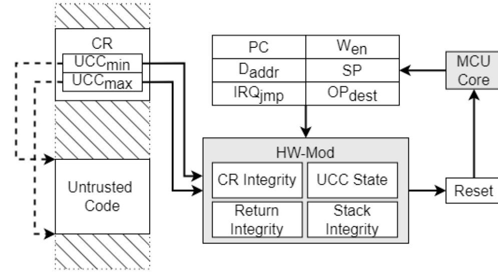


Fig. 1. **UCCA** hardware architecture illustrating one *UCC*.

TABLE I
UCCA NOTATION

Notation	Description
UCC	Untrusted Code Compartment: An untrusted memory region
UCC_{min}	The address of the first instruction of *UCC*
UCC_{max}	The address of the final instruction of *UCC*
CR	Configuration Region: Protected memory region that stores UCC_{min} and UCC_{max} for each *UCC*
PC	The current value of the Program Counter
D_{addr}	The memory address accessed by an MCU memory access
W_{en}	A 1-bit signal set when the MCU is writing to memory
SP	The memory address of the current top of the stack
IRQ_{jmp}	A 1-bit signal set if a jump to an interrupt is occurring
ISR	An Interrupt Service Handler executed for a given interrupt
OP_{ret}	The return address of call, interrupt, and exec instructions
reset	A 1-bit signal indicating a violation occurred and resetting the MCU
$reset_{ucca}$	A copy of the *reset* signal used by each sub-module
RET_{exp}	The expected return address of *UCC* saved by *UCCA*
BP	The address of the bottom of *UCC*'s isolated stack frame

1) The program counter (PC), containing the address of the currently executing instruction.
2) The data address access signal (D_{addr}), containing the memory address accessed by the current instruction (if any).
3) The write enable bit (W_{en}), indicating if the current memory access (if any), is a write access.
4) The stack pointer (SP), indicating the memory address of the last data element added to the stack.
5) The interrupt jump bit (IRQ_{jmp}), indicating if a jump to an ISR is occurring.
6) The operation return (OP_{ret}), containing the return address saved when call, interrupt, or exec instructions occur.

If a violation of *UCCA* properties occurs, a 1-bit *reset* output signal is set. This signal resets the MCU core immediately, i.e., before executing the following instruction. As noted earlier, we treat violations with resets for simplicity but software-based exception handling is also possible. HW-Mod runs in parallel with the MCU core to monitor these values for each executed instruction. Table I summarizes these signals and the notation used in the remainder of this article.

HW-Mod is composed of multiple submodules that enforce different *UCCA* properties. The *CR* Integrity submodule protects *CR* (which stores *UCC* definitions) from being overwritten at run-time. The Return Integrity submodule enforces correct returns from *UCC*s. The Stack Integrity submodule prevents a *UCC* from corrupting the stack pointer or overwriting external data in the MCU stack. Finally, the UCC State submodule determines whether a *UCC* is executing. This state is used by the Return and Stack Integrity submodules.

A dedicated instance of the UCC State, Return Integrity, and Stack Integrity submodules is required for each isolated *UCC*.

C. *UCCA Versus Existing Hardware*

As discussed in Section I, some MCUs have MPU support to protect memory regions. Therefore, a natural path to obtain untrusted code compartmentalization is with this existing support. Current MPUs enable the configuration of read, write, and execute permissions for up to 16 physical memory regions [40]. These permissions are further split for privileged and unprivileged software, however, unprivileged code cannot have more permissions than privileged code [15], [40].

To isolate untrusted code, the MPU must first separate the untrusted code from the rest of the program. This can be done by setting the untrusted code as unprivileged and the remainder of the binary as privileged. Then, the privileged code can be marked executable in privileged mode while the unprivileged (untrusted) code is executable in both contexts. This allows the application to freely call the untrusted code but prevents the untrusted code from jumping back into the rest of the binary. However, this does not prevent untrusted code from accessing other untrusted regions. As all untrusted code is unprivileged and executable by unprivileged code, independent untrusted segments can freely call each other, preventing isolation between untrusted regions. Similarly, the remainder of the application is now privileged. As privileged code can overwrite the MPU (and other system-level) configurations, this greatly increases the system's TCB. Also as the MPU only supports two privilege levels, isolating untrusted code prevents the MPU from isolating security-critical system code from applications in general.

While this model achieves isolation of untrusted code, it also prevents its execution outright. Since, untrusted code is unprivileged it cannot jump back into the now privileged application, thus execution cannot return from the untrusted region. Remedying this requires an "exit region" to handle these transitions. This privileged region needs to be executable to unprivileged (untrusted) code and all unprivileged return instructions must be instrumented to jump to the exit region. The exit region must also enforce return integrity. However, this requires saving the return address when calling untrusted code. As such, all branch instructions that could call untrusted code (including all dynamic branches) must also be instrumented.

For stack integrity, the MPU must define another region around the current stack when entering an untrusted region and mark it as read-only to unprivileged code. Moreover, the MPU would also need to maintain a shadow stack [41], [42] of return addresses and protected stack definitions otherwise when untrusted regions call each other, the current return address and protected stack region would be overwritten, re-exposing the system to an attack.

Due to these requirements, implementing a single MPU *UCC* would require at least four MPU regions. It also requires heavy binary instrumentation and dynamic MPU reconfiguration leading to increased run-time overheads. Isolating multiple regions further requires the implementation of a shadow stack. MPU-based *UCC*s would also require disabling interrupts when executing *UCC*-resident code. Otherwise, $\mathcal{A}\mathsf{dv}$ could leverage interrupts to break isolation as they are privileged [43]. Thus, MPU-based untrusted code isolation results in large run-time and storage overheads as well as precludes applications' real-time response to asynchronous events.

One could also attempt to port TrustZone controls into an untrusted code isolation mechanism. However, similar to the MPU case, this would also have many limitations. A TrustZone-based implementation would require all untrusted code to be in the Nonsecure world, while the rest of the application would execute in the Secure world. This would greatly increase the Secure world TCB. Similarly, this configuration would prevent TrustZone from isolating security-critical code from the rest of the application. Again similar to MPU, TrustZone cannot mutually isolate different untrusted code sections alone. Instead, TrustZone-equipped MCUs often work alongside an MPU to provide further separation within each world. However, this requires the MPU be reconfigured between worlds, increasing the system's run-time overhead. Similarly, all calls to and returns from (including interrupts) untrusted code will require execution to change worlds. This requires a context switch where the Secure world's state is saved/restored and the MPU configuration is updated before execution continues. Along with this, any Secure world data passed to untrusted code must be marshaled (copied) to the Nonsecure world and any results must be marshaled back. All this saving, copying, and configuring greatly increases the run-time overhead of the system.

IV. **UCCA** DETAILS: FORMAL SPECIFICATION AND VERIFIED IMPLEMENTATION

We now discuss *UCCA* in detail. Our discussion focuses on a single *UCC* as multiple *UCC*s are obtained by simply instantiating multiple units of the same hardware modules (one per additional *UCC*). To formally verify *UCCA*'s implementation, we formalize each of *UCCA*'s security properties using LTL. We then design FSMs to enforce these requirements. The individual FSMs are implemented in Verilog HDL and combined into one Verilog design for HW-Mod (as shown in Fig. 1). Finally, HW-Mod and each submodule are automatically translated to the SMV model checking language [44], using Verilog2SMV [28]. The resulting SMV models are checked against all required LTL specifications, using the NuSMV model checker [27], to produce a proof of the correctness of *UCCA*'s implementation with respect to the LTL.

UCCA modules are implemented as mealy FSMs (where outputs change with the current state and current inputs). Each FSM has one output: a local *reset*. *UCCA*'s output *reset* is given by the disjunction (logic *or*) of the local *reset*-s of all submodules. Thus, a violation detected by any submodule causes *UCCA* to trigger an immediate MCU reset. To ease presentation, we do not explicitly represent the value of the *reset* output in our FSMs. Instead, we define the following implicit representation:

1) *reset* is 1 whenever an FSM transitions to the *Reset* state;
2) *reset* remains 1 until transitioning out of the *Reset* state;
3) *reset* is 0 in all the other states.

$$\mathbf{G}:\{reset \implies [(\neg(PC \in UCC) \wedge (\mathbf{X}(PC) \in UCC) \implies (\mathbf{X}(RET_{exp}) = OP_{ret}) \vee reset)\mathbf{W}(PC \in UCC)]\} \tag{2}$$

$$\mathbf{G}:\{\neg(PC \in UCC) \wedge (\mathbf{Y}(PC) \in UCC) \wedge \neg\mathbf{Y}(IRQ_{jmp}) \implies [(\mathbf{X}(PC) \in UCC \implies (\mathbf{X}(RET_{exp}) = OP_{ret}) \vee reset)\mathbf{W}(PC \in UCC)]\} \tag{3}$$

$$\mathbf{G}:\{(PC \in UCC) \wedge \neg(\mathbf{Y}(PC) \in UCC) \implies [((\mathbf{X}(RET_{exp}) = RET_{exp}) \vee reset)\mathbf{W}(\neg(PC \in UCC))]\} \tag{4}$$

$$\mathbf{G}:\{\neg(PC \in UCC) \wedge (\mathbf{Y}(PC) \in UCC) \wedge \mathbf{Y}(IRQ_{jmp}) \wedge \neg\mathbf{Y}(reset) \implies [(\mathbf{X}(RET_{exp}) = RET_{exp})\mathbf{W}((PC \in UCC) \vee reset)]\} \tag{5}$$

$$\mathbf{G}:\{\neg reset \wedge (PC \in UCC) \wedge \neg(\mathbf{X}(PC) \in UCC) \wedge \neg IRQ_{jmp} \implies (\mathbf{X}(PC) = RET_{exp}) \vee \mathbf{X}(reset)\} \tag{6}$$

Fig. 2. Return integrity module LTL specifications.

Note that, all FSMs remain in the *Reset* state until $PC = 0$, which signals that the MCU reset routine is finished.

A. Defining Isolated UCCs

Each *UCC* is defined by the first and last addresses of its code: UCC_{min} and UCC_{max}, respectively. They mark the untrusted executable's location in memory. While *UCC* can have arbitrary size, the smallest unit of code *UCCA* can isolate is a single function, where UCC_{min} and UCC_{max} are the addresses of the first and last instruction in the function, respectively. Attempts to isolate smaller regions (i.e., partial functions) would result in return integrity violations. Also, *UCC*s should not partially overlap, since each *UCC* is an independent code section. As such, partially overlapping regions would again cause return integrity violations. While partially overlapping *UCC*s are invalid, *UCCA* allows nested *UCC*s. Nested *UCC*s support different levels of distrust within an untrusted compartment, further constraining vulnerabilities within the inner *UCC* from spreading to the outer region. Similarly, each *UCC* must be self-contained, i.e., include the untrusted code and its dependencies (such as callback implementations it relies upon). All other/trusted code should remain outside *UCC* limiting its exposure to the potentially vulnerable code within *UCC*.

B. Integrity of UCC Boundaries

UCC_{min} and UCC_{max} can vary depending on the untrusted executable being compartmentalized. During cross-compilation/linking, appropriate *UCC* values are determined and stored in *CR* at load time. At run-time, *UCCA* uses the values stored in *CR* to monitor the execution of *UCC*-resident code. To prevent $\mathcal{A}\mathsf{dv}$ from altering UCC_{min} and UCC_{max} at run-time (effectively disabling *UCCA*), *UCCA*'s *CR* integrity submodule ensures *CR* is immutable. *CR* integrity is defined in LTL specification 1 which states that at all times (**G** LTL quantifier) *UCCA* sets $reset = 1$ if an attempt to write to *CR* is detected. Attempts to write to *CR* are captured by checking if the W_{en} bit is set while the D_{addr} signal points to a location within *CR* reserved memory. This ensures that *UCC* definitions cannot be changed at run-time

$$\mathbf{G} : \{[(D_{\text{addr}} \in CR) \wedge W_{en}] \implies \text{reset}\}. \tag{1}$$

The CR integrity FSM is formally verified to adhere to LTL specification 1. Due to its relative simplicity, we do not visualize the FSM. The FSM has two states: *Run* and *Reset*. The *Run* state represents the MCU's normal operation. If an attempt to write to *CR* is detected the state transitions to *Reset*. The FSM remains in this state until the reset process has been completed (indicated by having $PC = 0$) at which point the FSM transitions back to the *Run* state.

C. Enforcing UCC Return Integrity

Return integrity prevents control flow attacks within *UCC* from escalating to the rest of the system by ensuring that *UCC* returns to the correct address (disallowing any jumps from within *UCC* to an invalid external location). Since, *UCC* has to isolate at least one function, execution must enter *UCC* through a call or interrupt (irq) instruction and leave through a return instruction. *UCCA* leverages this behavior to provide return integrity, by saving the correct return address internally (RET_{exp}) when *UCC* is called. Then, when execution returns from *UCC*, *UCCA* checks that the actual return address matches RET_{exp}. Fig. 2 depicts the LTL specifications defined to enforce return integrity.

UCCA saves the return address rather than protecting its value on the stack as return instructions assume that the return address is at the top of the stack when called. In benign circumstances, this holds as data on the stack is freed ("popped") before a return. However, as *UCC* is assumed to be vulnerable, execution can jump directly to a return instruction bypassing the required "pops." Thus, protecting the return address alone would not prevent this type of attack.

To check that *UCC* returns to the correct location, *UCCA* must first save the correct return address. LTLs 2 and 3 specify how RET_{exp} is saved. Both statements stipulate that when execution enters *UCC*, *UCCA* sets RET_{exp} to the correct return address (OP_{ret}) otherwise the device is in an invalid state (*reset*). Whether the execution is entering *UCC* is determined by the current and next PC values. The next value of PC is represented using the LTL ne**X**t operator X(PC). If the current value of PC is outside *UCC* and X(PC) is within *UCC*, execution is entering *UCC*. OP_{ret} is the correct return address as OP_{ret} is the return address written to the stack by the MCU core. Both statements are also conditioned on W($PC \in UCC$). This states that this RET_{exp} saving behavior is true until execution enters *UCC* (or always true should execution never enter *UCC*). In other words, this behavior is only true for the next execution of *UCC*. While both specifications are similar, LTL 3 states that when *UCC* finishes executing, the correct return address is saved the next time execution enters *UCC*. Whether the execution of *UCC* is finished is determined by the current and previous values of PC and the previous value of IRQ_{jmp}. Previous values are represented using the LTL **Y**esterday operator (i.e., Y(PC)). The IRQ_{jmp} signal indicates if a jump to an ISR is occurring. If PC was previously within *UCC* and is now outside *UCC*, execution has left *UCC*. If execution left *UCC* (and this was not due to

an interrupt: $\neg Y(\text{IRQ}_{jmp})$), then *UCC* has finished executing. Since, this statement conditions the next execution of *UCC* on the previous iteration, it guarantees that the correct return address is saved every time *UCC* is called, except for its first execution. Instead, LTL 2 ensures the proper return address is saved for this initial execution. LTL 2 states that after a device reset, the next time execution enters *UCC*, OP_{ret} is saved to RET_{exp}. *UCCA* always initializes in a reset condition. As such, at boot, this statement also applies. Taken together, LTL statements 2 and 3 ensure that RET_{exp} stores the correct value whenever *UCC* is called.

Once saved, RET_{exp} must remain fixed until *UCC* finishes executing to ensure that return integrity only allows valid return addresses. Thus, RET_{exp} is immutable while executing *UCC*. This property is defined in LTL 4. Entrance to *UCC* is again determined using the current and previous value of PC. If PC is currently within *UCC* and the previous value was outside *UCC*, then execution has just entered *UCC*. RET_{exp}'s immutability is captured by checking that the current value of RET_{exp} always matches the next ($\text{X}(\text{RET}_{\text{exp}})$) while within *UCC* ($\text{W}(\neg(PC \in UCC))$). However, *UCC* is interruptable so to ensure that RET_{exp} remains correct, RET_{exp} must also be immutable across interrupts. LTL 5 describes this behavior and states that, when execution leaves *UCC* due to an interrupt and the device is not resetting ($\neg Y(reset)$), RET_{exp} is immutable until execution of *UCC* resumes, or until a device reset occurs. Added together these two specifications ensure that once execution of *UCC* begins, RET_{exp} cannot change until it finishes or the device resets.

Finally, return integrity is described in LTL 6. This specification states that when execution exits *UCC* (not due to an interrupt), an exception (reset) is triggered unless the actual and saved return addresses match. Unlike specifications 3 and 5, exiting a region is detected using the current and next value of PC. Specifically, execution is exiting the region if PC is currently in *UCC* and the X(PC) is outside *UCC*. Due to this, X(PC) is the actual value of the return address. Therefore, *UCCA* compares RET_{exp} to X(PC) and sets $reset = 1$ if a violation is detected.

Fig. 3 depicts the Verilog FSM implemented by the return integrity submodule and formally verified to simultaneously adhere to LTL specifications 2–6. The FSM defines four states: 1) *Out*, 2) *In*, 3) *IRQ*, and 5) *Reset*. *Out* represents when PC is outside of *UCC*. Once execution enters *UCC* ($PC \in UCC$), the FSM transitions to *In*. While executing *UCC*, the FSM remains in the *In* state. If an interrupt occurs while within *UCC*, the FSM transitions to the *IRQ* state. If execution has just entered *UCC* when an interrupt occurs, it is also possible for *Out* to transition directly to the *IRQ* state. *IRQ* represents when *UCC* has been interrupted. While in *IRQ*, RET_{exp} is maintained. *IRQ* transitions back to the *In* state once *UCC* resumes. When the execution leaves *UCC* ($\neg(PC \in \text{UCC})$) (not due to an interrupt), if execution returns to the expected memory address ($PC = \text{RET}_{\text{exp}}$) it is a valid return and the FSM transitions to the *Out* state. Otherwise, a violation of return integrity has occurred and the FSM transitions to the *Reset* state. Once the reset routine is completed, the FSM transitions to the *Out* state. For synchronization, *Out*, *In*, and *IRQ* also transition

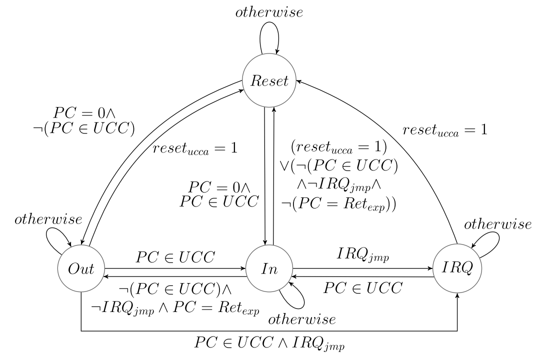


Fig. 3. Verified FSM for return integrity.

to *Reset* if a violation occurs in another module or *UCC* ($\text{reset}_{ucca} = 1$).

D. UCC Entry and Exit Points

Despite being untrusted, *UCCA* allows execution to enter and exit *UCC* at/from any instruction in the region. This is because *UCCA* prevents attacks within *UCC* from escalating to the remainder of the system. To that end, in terms of control flow integrity (CFI), it suffices to ensure that the *UCC* caller code resumes correctly. As *UCC*-resident code is untrusted (e.g., third party libraries), *UCCA* does not enforce properties regarding its internal behavior. By allowing arbitrary entry and exit points, multiple functions can be isolated by a single *UCC* and all remain directly callable by external code.

E. Protecting Stack Data Outside UCC's Frame

Return integrity prevents escalation of attacks, such as control flow hijacking and ROP. However, *UCC*-resident code may still attempt to escalate data-flow attacks [45], [46], [47] that overwrite data on the stack or create a malicious stack. Editing the stack has no immediate effect on a program's control flow. Therefore, return integrity is not violated. However, as a program's behavior depends on its variables, editing stack data could still compromise execution integrity.

To prevent data-flow attacks, *UCCA* creates an isolated stack frame for *UCC*. Stack frames are a memory management technique that segments the stack into different sections corresponding to different function calls [6]. To define a frame, *UCCA* stores the initial stack pointer (SP) of the previous instruction when entering *UCC*. Since, execution enters *UCC* through either a call or interrupt, *UCCA* saves SP before the return address is pushed to the stack. We use this value for the base of the *UCC*'s frame for multiple reasons. First, this value separates non-*UCC* and *UCC* data. As no *UCC*-resident code has been executed yet, all *UCC* data will be written above this value. Second, this value is what SP should be when execution returns from *UCC*. When exiting *UCC*, the return instruction removes the return address from the stack. Thus, upon exit, SP should be the same value as before the call to *UCC*. We refer to the saved SP value as the base pointer (*BP*) in the remainder of this article. To isolate *UCC*'s frame, *UCCA* blocks all the write attempts performed by *UCC*-resident code to addresses below *BP* and enforces the proper stack context ($SP = BP$) when exiting *UCC*.

$$\mathbf{G}:\{reset \implies [\neg(\mathbf{Y}(PC) = PC) \implies (BP = \mathbf{Y}(SP)) \lor reset)\mathbf{W}(PC \in UCC)]\} \quad (7)$$
$$\mathbf{G}:\{\neg(PC \in UCC) \land (\mathbf{X}(PC) \in UCC) \implies (\mathbf{X}(BP) = BP) \lor reset\} \quad (8)$$
$$\mathbf{G}:\{\neg(PC \in UCC) \land (\mathbf{Y}(PC) \in UCC) \land \neg\mathbf{Y}(IRQ_{jmp}) \implies [(\neg(\mathbf{Y}(PC) = PC) \implies (BP = \mathbf{Y}(SP)) \lor reset)\mathbf{W}(PC \in UCC)]\} \quad (9)$$
$$\mathbf{G}:\{(PC \in UCC) \land \neg(\mathbf{Y}(PC) \in UCC) \implies [((\mathbf{X}(BP) = BP) \lor reset)\mathbf{W}(\neg(PC \in UCC))]\} \quad (10)$$
$$\mathbf{G}:\{\neg(PC \in UCC) \land (\mathbf{Y}(PC) \in UCC) \land \mathbf{Y}(IRQ_{jmp}) \land \neg\mathbf{Y}(reset) \implies [(\mathbf{X}(BP) = BP)\mathbf{W}((PC \in UCC) \lor reset)]\} \quad (11)$$
$$\mathbf{G}:\{[(PC \in UCC) \land W_{en} \land (D_{addr} \geq BP)] \implies reset\} \quad (12)$$
$$\mathbf{G}:\{\neg reset \land (PC \in UCC) \land \neg(\mathbf{X}(PC) \in UCC) \land \neg IRQ_{jmp} \implies (\mathbf{X}(SP) = BP) \lor \mathbf{X}(reset)\} \quad (13)$$

Fig. 4. Stack integrity module LTL specifications.

Consequently, writes to stack variables passed by reference into *UCC* are also blocked as they result in writes below *BP*. Instead, as the heap is above the stack [48] (and thus *BP*) data passed by reference to *UCC* should first be copied to the heap. Then, when execution returns from *UCC*, the edited heap value can be copied back to the original. We emphasize that (contrary to writes) the stack is always readable from *UCC*. Thus, this marshaling is only necessary for pre-existing stack data that is meant to be written by code within a *UCC*. Global variables are also stored above the stack by default in the target architecture [48], [49] and thus writable by *UCC*-resident code. This is expected as the global variables are meant to be accessible to the whole program. Nonetheless, if desired, selected global data can be linked (at compile-time) to appear below the stack, preventing writes from *UCC*.

Fig. 4 lists the LTL statements defined to enforce stack integrity. LTL 7 states that after a reset, whenever the executing instruction changes ($\neg(Y(PC) = PC)$) *BP* contains the previous value of SP ($BP = Y(SP)$) until execution enters *UCC*. While *BP* saves the previous SP, this actually represents the initial SP for the current instruction. Thus, when *UCC* is called, *BP* holds the value of SP before the return address is pushed to the stack. By conditioning on a reset, this statement ensures that *BP* is correct when calling *UCC* for the first time. LTL 9 similarly states that when *UCC* finishes executing, *BP* stores the previous value of SP whenever the current instruction changes until execution re-enters *UCC*. This rule ensures the BP is also correct on all the subsequent executions of *UCC*. Once *UCC* is running, LTLs 10 and 11 ensure *BP* cannot be changed. LTL 10 states that when execution enters *UCC*, *BP* is immutable until execution leaves *UCC*. LTL 11 states that if *UCC* is interrupted, *BP* is immutable until *UCC* resumes or a reset occurs. Together these statements ensure that once in *UCC*, *BP* cannot be changed until the execution of *UCC* completes. However, the value of *BP* is ambiguous when execution enters *UCC*. At this instance, LTLs 7 and 9 do not hold, but, LTL 10 only holds from this point forward. Thus, to ensure *BP* is still correct LTL 8 states that when execution is entering *UCC*, *BP* does not change ($X(BP) = BP$). Combined with LTLs 7 and 9, these statements ensure that *BP* is properly set whenever execution enters *UCC*.

UCCA's stack frame isolation is defined in LTL specification 12. This specification states that, at all times, *UCCA* sets $reset = 1$ if execution is within *UCC* and attempts to write to the stack outside its stack frame. Writes outside the isolated frame are captured by the W_{en} bit being set while the D_{addr} signal points to a location below *BP*. D_{addr} is below *BP* if $D_{\text{addr}} \geq BP$ as the stack grows toward 0. Hence, values below *BP* have a larger address than *BP*. Stack isolation ensures that the *UCC*-resident code cannot tamper with data memory in use by the remainder of the system.

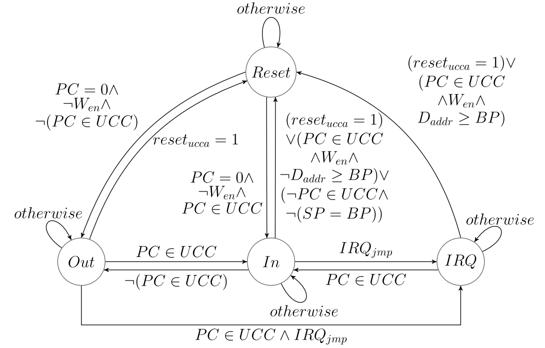


Fig. 5. Verified FSM for stack integrity.

Finally, LTL 13 ensures that the stack pointer is properly restored before execution leaves *UCC*. It states that, if the device is not already resetting and execution is leaving *UCC* (not due to an interrupt), the next SP should be *BP* ($X(SP) = BP$). Since, *BP* represents the value of SP at the start of the call to *UCC*, this check enforces that SP returns to the same value as before executing *UCC*. This prevents an adversary from corrupting SP such that malicious data written to the stack by *UCC* resident code is used by non-*UCC* code.

Fig. 5 depicts the Verilog FSM implemented by the stack integrity module and formally verified to adhere to LTL specifications 7–13. The FSM defines four states: 1) *Out*, 2) *In*, 3) *IRQ*, and 4) *Reset*. The stack integrity FSM behaves similarly to the return integrity FSM with a few exceptions. First, when in the *IRQ* state, *BP* is maintained until the execution of *UCC* is resumed rather than RET_{exp}. Similarly, when transitioning to *Out*, SP must equal to *BP* otherwise the FSM transitions to the *Reset* state. Finally, while in *UCC*, any write below *BP* will violate the stack isolation and cause the FSM to transition to the *Reset* state.

V. Security Analysis

Recall from Section III-A that $\mathcal{A}$dv aims to escalate vulnerabilities located within *UCC*s to compromise the remainder of the system with attacks, such as control flow hijacks, ROP, data corruption, and code injection. In this section, we argue that such attempts are unsuccessful due to *UCCA* guarantees.

$\mathcal{A}$dv may try to leverage vulnerabilities to alter the control flow of the binary and jump to an arbitrary location in memory. For this, $\mathcal{A}$dv would need to exploit a branching instruction, such as a return, within a *UCC*. $\mathcal{A}$dv would either need to overwrite a return/jump address on the stack or cause data on the stack to be misinterpreted as an address. Using vulnerabilities within *UCC*, $\mathcal{A}$dv could attempt to hijack an intermediate instruction or the final return instruction to jump to an arbitrary address. However, this malicious jump would not match the saved return address (LTLs 2–5) and the attack would be stopped (LTL 6).

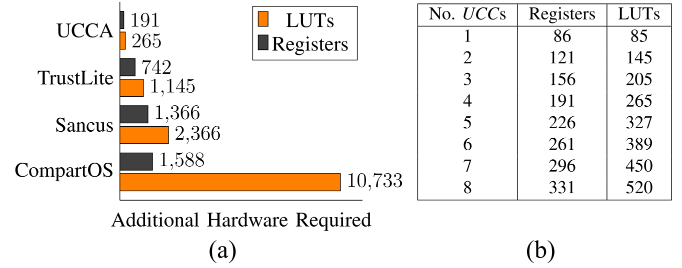
$\mathcal{A}$dv could also attempt to overwrite code in program memory or data on the stack. Both scenarios would allow $\mathcal{A}$dv to alter program behavior outside *UCC*s. Program memory is located below the stack, thus always outside *UCC*'s isolated stack frame. Similarly, all non-*UCC* data falls below its frame's *BP*, and is outside *UCC*'s isolated frame. As such, $\mathcal{A}$dv cannot overwrite code in program memory and non-*UCC* data on the stack (LTLs 7–12). $\mathcal{A}$dv could also attempt to write malicious data to the stack and corrupt SP such that the device uses the malicious stack once execution leaves *UCC*. However, this would require SP not be equal to *BP* when leaving *UCC* which is prevented by stack integrity (LTLs 7–11 and 13). Finally, $\mathcal{A}$dv could attempt to inject and execute code on the stack or heap, however *UCCA* prevents this as executing data memory requires execution to leave *UCC* violating return integrity (LTLs 2–6).

Interrupts can bypass the isolation enforced by *UCCA*. As such, $\mathcal{A}$dv may try to abuse this behavior and exploit an interrupt to escape *UCCA*'s restriction. However, similar to any untrusted code in *UCCA*, if an ISR is untrusted, it can also be defined as a *UCC*. As a consequence, since the untrusted interrupt is isolated, return and stack integrity prevent it from escalating to the remainder of the system (LTLs 2–13). $\mathcal{A}$dv could also attempt to overwrite the address of an ISR in the Interrupt Vector Table (IVT). This would cause execution to jump to an $\mathcal{A}$dv defined value, whenever the corrupted interrupt is triggered. Similar to program memory, IVT is stored below the stack. As such it is always outside *UCC*'s isolated stack frame and not writable by $\mathcal{A}$dv (LTLs 7–12).

Finally, $\mathcal{A}$dv may attempt to disable *UCCA* and break isolation by overwriting *UCC* region definitions stored in *CR*. However, *CR* is immutable at run-time (LTL 1). The only way to overwrite *CR* is by physically reprogramming the MCU which contradicts the $\mathcal{A}$dv model.

VI. Prototype and Evaluation

We implemented *UCCA* on the OpenMSP430 core [48]. *UCCA* realizes the hardware architecture depicted in Fig. 1. Along with HW-Mod, we implement a simple peripheral module for *CR*. The peripheral module allows for *UCC* definitions to be stored and accessed by HW-Mod at a predefined fixed data memory location. We use Xilinx Vivado [50] to synthesize an RTL prototype of *UCCA* in real hardware. *UCCA*'s design was deployed on a Basys-3 prototyping board [51], that features an Artix-7 commodity FPGA [52]. Our implementation is available at [20].



No. *UCC*s	Registers	LUTs
1	86	85
2	121	145
3	156	205
4	191	265
5	226	327
6	261	389
7	296	450
8	331	520

Fig. 6. *UCCA* Evaluation: (a) HW cost comparison with 4 *UCC*s; (b) Added HW by total *UCC*s.

A. *UCCA Evaluation*

TCB Size: To calculate *UCCA*'s TCB size we count the amount of Verilog code needed to implement HW-Mod. Since, *UCCA* was implemented in hardware and works independently from the MCU core, *UCCA*'s TCB only consists of HW-Mod. The *UCCA* prototype with support for one *UCC* was implemented using *423 lines of Verilog code*. Each additional *UCC* supported by *UCCA* adds another 21 lines of Verilog to the TCB, for instantiating the same modules repeatedly.

Hardware and Memory Overhead: The number of required *UCC*s is application dependent. Due to this, we measure *UCCA* considering support from one to eight *UCC*s and estimate the cost for arbitrarily many *UCC*s. The additional hardware cost is calculated by looking at the number of added Look-Up Tables (LUTs) and Registers. The increase in the number of LUTs is an estimate of the additional chip cost and size required for combinatorial logic, while the number of registers offers an estimate of the state overhead required by the sequential logic in *UCCA* FSMs. A summary of the hardware cost is shown in Fig. 6(b). To isolate a single *UCC*, *UCCA* requires an additional 86 registers and 85 LUTs. This constitutes a respective 12.4% and 4.7% increase in registers and LUTs atop the unmodified OpenMSP430 core. In the largest test, with 8 *UCC*s, *UCCA* added 331 registers and 520 LUTs to the underlying system. This equates to a 47.8% and 29% increase in registers and LUTs.

In general, *UCCA* can support arbitrarily many *UCC*s with the only limiting factor being the additional hardware cost per region. We can predict the overhead for any *UCCA* configuration as the overhead grows linearly with the number of *UCC*s. As previously stated, *UCCA* with one *UCC* adds 86 registers to the MCU. However, each subsequent *UCC* added only requires an additional 35 registers. Similarly, *UCCA* with one *UCC* adds an initial 85 LUTs to the MCU. Each additional *UCC* adds on average 62 LUTs to the system (variance is due to the synthesis tool heuristic). Thus, *UCCA* with support for *N* *UCC*s can be estimated as

$$\text{LUTs} \approxeq 62 \times (N - 1) + 85 \tag{14}$$

$$\text{Registers} = 35 \times (N - 1) + 86. \tag{15}$$

UCCA also introduces a small storage overhead. Each *UCC*'s $UCC_{\min}$ and $UCC_{\max}$ are stored in *CR* in the device's peripheral memory. Each address is 2 bytes long so each *UCC* requires 4 bytes of data memory. On the OpenMSP430, peripheral memory can be between 512B and 32KB long [48].

Thus, each *UCC* incurs between 0.01% and 0.78% memory overhead depending on the size of peripheral memory.

Energy Overhead: To evaluate the energy consumption caused by *UCCA* added hardware, similar to prior work [22], [53], [54], we use the Vivado synthesis tool [50] to estimate *UCCA*'s power consumption on our FPGA prototype. We consider *UCCA* with support for 8 *UCC*s. In this configuration, the MCU consumes 69 mW of static power with *UCCA* accounting for 1 mW (1.45%) of the total static consumption. The dynamic power consumption depends on how frequently *UCCA*'s internal registers are updated. We evaluate *UCCA* on an application that loops through multiple function calls that modify the stack. We consider this a worst-case as it causes each *UCC*s' internal RET_{exp} and *BP* to update constantly. Running this application resulted in a total dynamic draw of 113 mW where *UCCA* accounted for 1 mW (0.88%) of this consumption. Doubling the number of *UCC*s to 16 increased the total dynamic draw to 114 mW. Thus, each *UCC* introduces $\approx$ 0.125 mW of dynamic power draw.

Run-Time Overhead: UCCA does not modify the MCU core or Instruction Set Architecture (ISA). HW-Mod performs *UCC*-related checks in parallel with the MCU core. These checks incur no extra run-time cycles to the software execution and do not interfere with the MCU's ability to respond to real-time events. As HW-Mod accesses memory through a dedicated physical channel, separate from the normal MCU core access channels, it does not cause interference or contention.

The only source of run-time overhead in *UCCA* is due to marshaling data inputs to be modified by *UCC*-resident code. In these cases, the data must be first copied to the designated heap region before calling *UCC*-resident code. While the copying is done before *UCC* execution, it affects the overall system run-time. The associated run-time depends on the amount of data to be copied. In our prototype (based on MSP430), copying a "word" (2 Bytes, in this 16-bit architecture) requires one execution cycle of the absolute `MOV` instruction. This number scales linearly with the amount of data to be copied, i.e., an additional `MOV` instruction cycle is required for each pair of Bytes to be copied.

Formal Verification: We verified *UCCA* on an Ubuntu 20.04 machine running at 3.70 GHz. Total verification time was about 11.5 min with maximum memory allocation of 125 MB, which is within the resources of commodity computers.

Test Applications: To demonstrate *UCCA* protections, we implemented multiple test applications which are also available and discussed in more detail in our public *UCCA* release [20].

B. Comparative Evaluation

We compare *UCCA*'s overhead with three related schemes: 1) Sancus [22], 2) TrustLite [21], and 3) CompartOS [23].

Sancus provides memory isolation and attestation for shared remote embedded systems. Sancus introduces the **protect** and **unprotect** hardware instructions to create (and destroy) isolated software modules. Isolation is enforced by defining a fixed entry point for each module and using the program counter to restrict access to a module's data to module resident code only. Sancus also enables key storage for each module to allow for remote attestation [55], [56] of the region.

TrustLite is another isolation architecture that isolates individual software tasks or trustlets. Trustlet definitions are recorded in the Trustlet Table in protected memory. For access control, TrustLite uses an execution aware MPU (EA-MPU) which extends the read, write, and execute permissions with the current value of the program counter. This allows the EA-MPU to restrict trustlet access to a predefined set of entry points and prevent access to trustlet data from outside the trustlet. The trustlets, Trustlet Table, and EA-MPU are all configured by a privileged process named the SecureLoader when the MCU boots.

CompartOS provides automatic software compartmentalization for high-end embedded systems. CompartOS uses the CHERI [57] hardware capability system for memory isolation. CHERI adds the capability data type and capability-aware instructions to the device's ISA. Capabilities extend integer pointers with metadata, including bounds, permissions, and a validity bit to assign explicit permissions to the code they reference. Capabilities can also be "sealed" to link code and data capabilities together and prevent their modification. CompartOS uses capabilities to define compartments and seals/unseals them to context switch between different compartments.

We note that, while these approaches use hardware to isolate MCU memory, they are not directly comparable to *UCCA*. None of the prior work focuses on isolating untrusted code sections, a feature unique to *UCCA*. Both CompartOS and TrustLite target larger devices than *UCCA*. *UCCA* is more comparable to Sancus as both were implemented on the OpenMSP430 architecture. However, Sancus performs remote attestation in addition to isolation. Despite these differences, we believe that such systems are the most closely related to *UCCA*. In our comparison, we consider default support for four isolated regions. The comparison is displayed in Fig. 6(a).

UCCA presents lower overhead. With support for four *UCC*s, it requires 13.9% of the registers and 12.1% of the LUTs required by Sancus for the same number of isolated regions. With support for eight *UCC*s, *UCCA* still only incurs about a fourth of the overhead (24.2% registers and 22% LUTs). *UCCA* performs similarly when compared to TrustLite. *UCCA* uses 25.7% of the registers and 23.1% of the LUTs TrustLite uses. At eight *UCC*s, *UCCA* still only uses 44.6% of registers and 45.4% of LUTs used by TrustLite.

When compared to CompartOS, *UCCA* uses 87.9% fewer registers and 97.5% fewer LUTs to isolate four compartments. However, unlike Sancus and TrustLite, whose overhead scales with the number of isolated regions, CompartOS has the same hardware overhead, regardless of how many regions it supports. As *UCCA* continues to isolate more regions, *UCCA*'s overhead will eventually surpass CompartOS's. However, these larger configurations are unlikely in low-end MCUs. Similarly, CompartOS uses 229% more registers and 598% more LUTs than the OpenMSP430 core itself. This overhead shows that CompartOS is impractical for such low-end MCUs.

VII. Extended Related Work

Aside from the techniques mentioned in Section I, there are several attempts to mitigate run-time vulnerabilities on MCUs.

CFI is a class of techniques that limit the destination of any control flow transfer to a set of valid addresses [33], [58], [59], [60]. We also include randomization techniques in this discussion [61], [62]. These approaches often use a control flow graph (CFG) or a directed graph of nodes representing atomic sections of a binary [63]. CFGs enable the enumeration of all paths through a program, however, as programs get more complex the enumeration becomes undecidable. Due to this, many schemes use imprecise approximations prone to false positives [1]. Other approaches focus solely on returns (notably, shadow stacks [42]) removing the need for path enumeration but incurring large hardware and/or software overheads [1].

MPU-based Compartmentalization segments a binary into separate regions of memory and enforces isolation between them. Many schemes, such as ACES [15] simply use existing MPU operations to provide stronger isolation by segmenting code and enforcing well defined entry points between them [14], [15], [64], [65], [66]. Other techniques extend MPU functionality by providing new isolation criteria [17], [21], [67]. For example, Toubkal [68] adds a new hardware monitor to restrict regions to specific hardware controllers.

ISA-based Compartmentalization adds new functionality to the MCU core itself rather than relying on hardware monitors [16], [22], [23], [69]. These controls introduce new hardware instructions to enable isolation [69], use the instruction pointer to validate memory accesses [22], and add new data types to the core [57]. ISA-based isolation requires access to the source code to recompile the binary with ISA-specific instructions. It also requires the CPU core and compiler to be trusted, increasing the system TCB and typically the hardware overhead.

VIII. Tradeoffs and Limitations

Fixed UCC Definitions and Total Number of UCCs: UCCA implements *UCC* definitions that are immutable at run-time. This enables *UCC*s within privileged code and ensures *UCCA* guarantees can not be disabled by any code at run-time. However, the total number of *UCC*s can be limiting in larger systems with more untrusted code sections to isolate. In these systems, either untrusted code must share regions or not all the untrusted code can be isolated. A tradeoff would be allowing *UCC* definitions to be configurable at run-time. This would allow for more flexibility and for *UCC*s to be reused by different code sections. However, it would introduce additional attack vectors and run-time overhead for switching the context between the *UCC*s. Alternatively, the future work could further optimize the per-*UCC* hardware cost in *UCCA*, so that more *UCC*s can be supported at the same cost.

Protecting Heap Data: By default, *UCCA* does not prevent *UCC*-resident code from accessing heap data. This design decision is based on the premise that many simple MCU applications avoid dynamic memory allocation for performance reasons. Nonetheless, in applications that require heap allocation, discretionary protection of heap data against *UCC*-code can be achieved by linking a portion of the heap to allocate below the stack. This new portion would be protected from the *UCC* modifications (similar to how global variables are treated in *UCCA*). A second unprotected portion of the heap could remain above the stack (where the modifications can be made by *UCC*s) and be used to share/marshal data into *UCC*s. This approach allows selected heap data to be writable to *UCC*s while protecting the remainder of the heap and the stack. It also requires no changes to the *UCCA* hardware architecture.

IX. Conclusion

We proposed *UCCA*: an architecture leveraging a formally verified hardware monitor to isolate untrusted code compartments (*UCC*s) and limit the scale of run-time attacks on MCUs. *UCC*s are configurable and have variable size, making *UCCA* compatible with different programs. Isolation of *UCC*s is enforced in hardware and cannot be disabled by compromised software. In addition, *UCCA* does not incur run-time overhead in terms of added CPU instructions/cycles. Similarly, *UCC*s remain interruptable maintaining support for real-time operations. *UCCA*'s security analysis demonstrates that, by enforcing return and stack integrity for *UCC*s, *UCCA* constrains software exploits to their origin. Our evaluation, based on an open-source and formally verified *UCCA* prototype, shows that *UCCA* incurs small hardware overhead.

Acknowledgment

The authors sincerely thank the EMSoft'24 anonymous reviewers for their constructive comments.

References

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proc. S&P*, 2013, pp. 48–62.
[2] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Security* , vol. 15, no. 1, pp. 1–34, 2012.
[3] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proc. CCS*, 2010, pp. 559–572.
[4] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proc. CCS*, 2011, pp. 30–40.
[5] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. CCS*, 2007, pp. 552–561.
[6] A. One, "Smashing the stack for fun and profit," *Phrack Mag.*, vol. 7, no. 49, pp. 14–16, 1996.
[7] N. P. Smith, "Stack smashing vulnerabilities in the UNIX operating system," 1997. [Online]. Available: https://web.eecs.umich.edu/˜aprakash/security/handouts/Stack_Smashing_Vulnerabilities_in_the_UNIX_Operating_System.pdf
[8] "CVE-2022-24796," 2022. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-24796
[9] "CVE-2021-34527," 2021. [Online]. Available: https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-34527
[10] T. Abera et al., "C-FLAT: Control-flow attestation for embedded systems software," in *Proc. CCS*, 2016, pp. 743–754.
[11] M. Antonakakis et al., "Understanding the Mirai botnet," in *Proc. USENIX Security*, 2017, pp. 1093–1110.
[12] N. Falliere et al., "W32. Stuxnet dossier," White Paper, Symantec Corp., Tempe, AZ, USA, vol. 5, no. 6, p. 29, 2011.

[13] (Arm, Cambridge, U.K.) *Arm® V8-M Architecture Reference Manual*. (Apr. 2022). [Online]. Available: https://developer.arm.com/documentation/ddi0553/bs/

[14] A. A. Clements et al., "Protecting bare-metal embedded systems with privilege overlays," in *Proc. S&P*, 2017, pp. 289–303.

[15] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "ACES: Automatic compartments for embedded systems," in *Proc. USENIX Security*, 2018, pp. 65–82.

[16] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μRAI: Securing embedded systems with return address integrity," in *Proc. NDSS*, 2020, pp. 1–18.

[17] A. Mera et al., "D-box: DMA-enabled compartmentalization for embedded applications," in *Proc. NDSS*, 2022, pp. 1–17.

[18] "Market leading RTOS (real time operating system) for embedded systems with Internet of Things extensions." 2024. [Online]. Available: https://www.freertos.org/

[19] O. Girard, "OpenMSP430," Jun. 2009. [Online]. Available: https://opencores.org/projects/openmsp430

[20] L. Tyler et al., "UCCA repository," 2024. [Online]. Available: https://github.com/RIT-CHAOS-SEC/UCCA

[21] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *Proc. EuroSys*, 2014, pp. 1–14.

[22] J. Noorman et al., "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *Proc. USENIX Security*, 2013, pp. 479–498.

[23] H. Almatary et al., "CompartOS: CHERI compartmentalization for embedded systems," 2022, *arXiv:2206.02852*.

[24] D. W. Loveland, *Automated Theorem Proving: A Logical Basis*. Amsterdam, The Netherlands: Elsevier, 2016.

[25] E. M. Clarke Jr., O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model Checking* (Cyber Physical Systems Series). Cambridge, MA, USA: MIT Press, 2018.

[26] D. Thomas and P. Moorby, *The Verilog® Hardware Description Language*. New York, NY, USA: Springer, 2008.

[27] A. Cimatti et al., "NuSMV 2: An opensource tool for symbolic model checking," in *Proc. CAV*, 2002, pp. 359–364.

[28] A. Irfan, A. Cimatti, A. Griggio, M. Roveri, and R. Sebastiani, "Verilog2SMV: A tool for word-level verification," in *Proc. DATE*, 2016, pp. 1156–1159.

[29] O. Lichtenstein, A. Pnueli, and L. Zuck, *The Glory of the Past*. Heidelberg, Germany: Springer, 1985.

[30] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *Proc. S&P*, 2015, pp. 745–762.

[31] I. Evans et al., "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proc. CCS*, 2015, pp. 901–913.

[32] Y. Younan, W. Joosen, and F. Piessens, "Runtime countermeasures for code injection attacks against C and C++ programs," *ACM Comput. Surveys*, vol. 44, no. 3, pp. 1–28, 2012.

[33] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *Proc. CCS*, 2008, pp. 15–26.

[34] "CVE-2017-0144," 2017. [Online]. Available: https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2017-0144

[35] "CVE-2022-42719," Oct. 2022. [Online]. Available: https://security-tracker.debian.org/tracker/CVE-2022-42719

[36] "CVE-2022-38040," Oct. 2022. [Online]. Available: https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2022-38040

[37] *ARM Security Technology—Building a Secure System Using TrustZone Technology*, ARM Ltd., Cambridge, U.K., 2009.

[38] (ARM Ltd., Cambridge, U.K.) *TrustZone Technology for Armv8-M Architecture Version 2.1*. (2019). [Online]. Available: https://developer.arm.com/documentation/100690/0201/

[39] S. Ravi, A. Raghunathan, and S. Chakradhar, "Tamper resistance mechanisms for secure embedded systems," in *Proc. VLSID*, 2004, pp. 605–611.

[40] (ARM Ltd., Cambridge, U.K.) *Chapter 5. Memory Protection Unit*. (2024). [Online]. Available: https://developer.arm.com/documentation/ddi0290/g/memory-protection-unit

[41] N. Burow, X. Zhang, and M. Payer, "SoK: Shining light on shadow stacks," in *Proc. S&P*, 2019, pp. 985–999.

[42] T. H. Y. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proc. CCS*, 2015, pp. 555–566.

[43] (ARM Ltd., Cambridge, U.K.) *Armv8-M Exception Model User Guide*. (2024). [Online]. Available: http://bit.ly/3SurBdL

[44] K. L. McMillan, "The SMV system," in *Symbolic Model Checking*. Boston, MA, USA: Springer, 1993, pp. 61–85.

[45] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *Proc. S&P*, 2016, pp. 969–986.

[46] S. Chen, J. Xu, and E. C. Sezer, "Non-control-data attacks are realistic threats," in *Proc. USENIX Security*, vol. 5, 2005, p. 146.

[47] N. Bellec, G. Hie, S. Rokicki, F. Tronel, and I. Puaut, "RT-DFI: Optimizing data-flow integrity for real-time systems," in *Proc. ECRTS*, 2022, pp. 18:1–18:24.

[48] O. Girard, "OPENMSP430," Nov. 2017. [Online]. Available: https://schaumont.dyn.wpi.edu/ece4530f19/pdf/openMSP430.pdf

[49] (Texas Instrum., Dallas, TX, USA). *MSP430F2xx, MSP430G2xx Family User's Guide*. (Aug. 2022). [Online]. Available: https://www.ti.com/lit/ug/slau144k/slau144k.pdf

[50] "Vivado design suite user guide: Using the Vivado IDE," 2022. [Online]. Available: http://www.pld.ttu.ee/–alsu/DD_Vivado.pdf

[51] "Basys 3 reference," 2024. [Online]. Available: https://digilent.com/reference/basys3/refmanual

[52] "Artix-7 FPGA family," 2024. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html

[53] A. Caulfield, N. Rattanavipanon, and I. De Oliveira Nunes, "ACFA: Secure runtime auditing & guaranteed device healing via active control flow attestation," in *Proc. USENIX Security*, 2023, pp. 5827–5844.

[54] A. J. Neto, A. Caulfield, C. Alvares, and I. De Oliveira Nunes, "DiCA: A hardware-software co-design for differential check-pointing in intermittently powered devices," in *Proc. ICCAD*, 2023, pp. 1–9.

[55] K. Eldefrawy, A. Francillon, D. Perito, and G. Tsudik, "Smart: Secure and minimal architecture for (establishing dynamic) root of trust," in *Proc. NDSS*, 2012, pp. 1–15.

[56] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A verified hardware/software co-design for remote attestation," in *Proc. USENIX Security*, 2019, pp. 1429–1446.

[57] R. N. M. Watson et al., "Capability hardware enhanced RISC instructions: CHERI instruction-set architecture (version 8)," Dept. Comput. Sci. Technol., Univ. Cambridge, Cambridge, U.K., Rep. UCAM-CL-TR-951 Oct. 2020. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.html

[58] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Security*, vol. 13, no. 1, pp. 1–40, 2009.

[59] G. Serra, P. Fara, G. Cicero, F. Restuccia, and A. Biondi, "PAC-PL: Enabling control-flow integrity with pointer authentication in FPGA SoC platforms," in *Proc. RTAS*, 2022, pp. 241–253.

[60] L. Davi et al., "HAFIX: Hardware-assisted flow integrity extension," in *Proc. DAC*, 2015, pp. 1–6.

[61] J. Shi, L. Guan, W. Li, D. Zhang, P. Chen, and N. Zhang, "HARM: Hardware-assisted continuous re-randomization for microcontrollers," in *Proc. EuroS&P*, 2022, pp. 520–536.

[62] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proc. CCS*, 2004, pp. 298–307.

[63] F. E. Allen, "Control flow analysis," *ACM SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[64] D. Kwon, J. Shin, G. Kim, B. Lee, Y. Cho, and Y. Paek, "uXOM: Efficient execute-only memory on ARMcortex-M," in *Proc. USENIX Security*, 2019, pp. 231–247.

[65] C. H. Kim et al., "Securing real-time microcontroller systems through customized memory view switching," in *Proc. NDSS*, 2018, pp. 1–15.

[66] A. Levy et al., "Multiprogramming a 64kB computer safely and efficiently," in *Proc. SOSP*, 2017, pp. 234–251.

[67] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny trust anchor for tiny devices," in *Proc. DAC*, 2015, pp. 1–6.

[68] A. Sensaoui, D. Hely, and O.-E.-K. Aktouf, "Toubkal: A flexible and efficient hardware isolation module for secure lightweight devices," in *Proc. EDCC*, 2019, pp. 31–38.

[69] R. Strackx, F. Piessens, and B. Preneel, "Efficient isolation of trusted subsystems in embedded systems," in *Proc. SecureComm*, 2010, pp. 344–361.