Secure Intermittent Computing with ARM TrustZone on the Cortex-M

Pietro Chiavassa
Politecnico di Torino
Turin, Italy
pietro.chiavassa@polito.it

Filippo Gandino
Politecnico di Torino
Turin, Italy
filippo.gandino@polito.it

Renato Ferrero
Politecnico di Torino
Turin, Italy
renato.ferrero@polito.it

Jan Tobias Mühlberg
Université Libre de Bruxelles
Brussels, Belgium
jan.tobias.muehlberg@ulb.be

***Abstract*—Computer systems that operate on volatile energy sources typically rely on *intermittent computing* approaches, which involve checkpointing the system's state and persisting a checkpoint to non-volatile memory before the system loses power, and then restoring this checkpointed state when the power supply becomes available again. This process allows for long-running tasks to make progress, but involves security risks when power interruption is used as an attack vector. Based on earlier work that secures checkpoints and checkpoint restoration on the MSP430 MCU, we implement and evaluate a secure intermittent computing protocol that relies on the security features of TrustZone on a Cortex-M MCU to protect the integrity, authenticity, state continuity, and freshness of checkpointed state. Our results show that checkpoints can be created or restored in 20–40 ms, depending on workload sizes. To the best of our knowledge, our work is the first to implement a complete checkpoint utility for the ARM TrustZone's secure world.**

1. Introduction

Intermittent computing is characterized by short periods of program execution that are interrupted by reboots, commonly due to power outages [21]. Intermittent computing (ImC) systems are often powered by energy harvesters, which extract energy from the environment using a variety of energy sources – e.g., solar, wind, vibration, or radio – and enable pervasiveness under a *deploy and forget* paradigm where no maintenance is required. Here, batteries are often insufficient as they have a limited lifetime and pose challenges regarding maintenance and disposal. Instead, ImC systems usually employ small energy buffers, capacitors or supercapacitors, determining their operating cycle: charge, operate, and die.

A challenge in ImC systems is to ensure forward progress of code execution, with the eventual completion of tasks. This requires systems to preserve information about the device state between power cycles and necessitates systems to track time, enable atomic execution of tasks, prevent wasted computation by correctly predicting remaining operation time and the duration of checkpointing, and avoid data inconsistencies due to premature interruption. Different techniques have been proposed to support this paradigm [21], [26], based either on hardware extensions, e.g., special memory or transistor technologies, or on software, e.g., ad-hoc compilers and specific programming techniques.

However, security aspects of ImC systems are often not considered. Vulnerabilities and attacks in ImC system can allow snooping, spoofing a replay of checkpoints, leading to a disruption of state continuity or to the retrieval of secret keys [17]. Specifically under the pervasive deployment paradigm, attackers may gain remote or physical access to unattended devices, e.g., read or replace storage elements. Still, many embedded platforms provide some basic security features, such as secure boot, memory protection, and cryptographic extensions to mitigate these challenges. Specifically, embedded Trusted Execution Environments (TEEs) [18] such as ARM TrustZone [12] could provide the necessary security primitives to implement dependable ImC, which we investigate in this paper.

Starting from the work on the Secure Intermittent Computing Protocol (SICP) for TI's MSP430FR5994 microcontroller [9], [10], this paper presents a secure ImC solution developed for the STM32U5 microcontroller. Differently from previous work [3], [9], [10], the entire security chain of the target platform is considered and no additional components beyond platform features, such as tamper-proof memory, are required. To the best of our knowledge, our work is the first to implement a complete checkpoint utility for ARM TrustZone's secure world.

Our target device comes with an ARM Cortex-M33 core, which features ARM TrustZone for Cortex-M. Besides protecting critical applications, the TEE features are used to provide strong isolation for a small Trusted Computing Base (TCB) that contains the secure checkpoint utility and its related data. Assuming that attackers can tamper with normal-world software but also with storage peripherals, we design a checkpointing approach that relies on the TEE's hardware keys to achieve state continuity and to protect checkpoint data so that we can store checkpoints on external FRAM. Our approach does not require additional hardware. Our paper makes the following contributions:

- We conceptualize a secure checkpoint utility for ARM Cortex-M, which we implemented for the STM32U5 MCU.
- We report on experiments with different approaches to storing checkpoints and assess the device's lifetime, finally selecting external SPI-connected Ferroelectric RAM (FRAM) as checkpoint storage.
- A SICP-based [10] security solution is designed considering the full security chain of the target platform, relying on TrustZone for Cortex-M.

- The performance and overheads of our secure checkpoint utility are carefully evaluated. We find that checkpoints can be created or restored in 20–40 ms, depending on workload sizes.
- Our prototype is provided open-source under an MIT license at https://github.com/ptrchv/STM32-IntermittentSecurity.

2. Background

ImC systems. Both hardware and software techniques are discussed in the literature to tackle the challenges of intermittent computing [21], [26].

On the hardware side, circuits are designed for efficient power supply and time tracking, e.g., using the discharge time of a dedicated capacitor. Memory architectures are designed to provide caches for non-volatile memories (NVMs), such as FRAM, STT-RAM, PLM, and RRAM. Other solutions concern non-volatile processors (NVPs) and application-specific integrated circuits (ASICs).

On the software side, the focus is on efficient programming and compiler techniques to preserve atomicity, avoid data inconsistencies, and reduce overhead when saving the device state. One common paradigm is splitting computation into atomic tasks that fit in the capacitor's charge. Task execution and checkpoint placement can be performed statically via compiler optimizations, or dynamically via timers or runtime checks of the remaining energy level. Efforts are also spent to reduce overhead for programmers and allow intermittent execution of unmodified legacy code. Approximate computing approaches deliberately introduce acceptable errors into the computing process of error-tolerant applications as a trade-off for energy-efficiency gains [25]. Deep neural network (DNN) compression is also explored.

MCU Platforms for Secure ImC. A limitation of custom hardware solutions is that they are not readily available on the market. Looking at off-the-shelf microcontrollers (MCUs), the TI's MSP430FR family is very common in ImC applications [21], since it features integrated FRAM memory. TI also provides the "compute through power loss" utility (CTPL) [19], which allows for saving CPU and peripheral states before entering a deep-sleep mode. Regarding security, the MSP430FR5994, which is used in the SICP studies [9], [10], features an AES co-processor, a fused JTAG interface to disable debug access, and a Memory Protection Unit (MPU) which also provides Intellectual Property (IP) encapsulation. IP encapsulation defines a memory segment where the contained data can only be accessed by code executed in the segment itself. However, commercial MSP430 processors do not support TEE functionality, with Sancus [15] being an FPGA-only research prototype that is not suitable to implement all aspects of an ImC system.

ARM TrustZone for Cortex-M. In this paper, we explore ARM TrustZone for Cortex-M [12], which is a TEE available in low-power MCUs. ARM TrustZone for Cortex-M provides hardware isolation between a secure (S) and a non-secure (NS) world, memory-based subdivision, where code residing in secure address ranges can access all the device's memory, while code executed from non-secure locations can only access non-secure memory [13].

The device boots from the secure world, and the security configuration is performed before jumping to non-secure. Then, the non-secure world can call secure services from specific entry points located in non-secure callable (NSC) memory. A reference implementation of secure services is provided by Trusted Firmware-M [20].

In traditional devices, memory isolation is usually handled by privileged code (e.g., a Real-Time OS) which manages the MPU. If an attacker gains access to privileged execution, the whole system is compromised. Instead, with TrustZone, access to privileged execution in a non-secure world cannot compromise secure services due to hardware isolation. This also has the benefit of reducing the Trusted Computing Base (TCB) to code residing in secure memory. Dynamic privilege-based memory isolation becomes an orthogonal concept in TrustZone system, where the MCU can be independently configured in each world [11].

3. Related Works

Checkpointing systems [5], [6], [16] for ImC and relative optimizations [1], [4], [8] are well discussed and evaluated in the related literature. However, security implications are commonly overlooked.

In [17], the authors present an attacker model where the attacker has physical access to the device (TI's MSP430FR5994) and can retrieve persistent data from memory, either via an unprotected JTAG interface, or via sophisticated probing techniques. Security vulnerabilities in CTPL are discussed, which allow snooping, spoofing, and replay of checkpoints. Finally, different attacks are performed against AES128 library of the device to retrieve the secret key.

These vulnerabilities are addressed by the Secure Intermittent Computing Protocol (SICP) [9], [10]. The protocol allows for authenticated checkpoints with multi-level confidentiality. It also manages the invalidation of previous checkpoints, to provide freshness against replay attacks, and atomicity, to prevent an invalid state of the system due to a power loss. The protocol only requires keeping the nonce and the secret keys in a small tamper-free NVM, since the attacker is assumed capable of reading and writing sections of the internal non-volatile memory. However, the device does not have tamper-free memory, which is only simulated using IP encapsulation features managed by the MPU.

In [3], TrustZone is used on a Cortex-M23 device to save arbitrary non-secure memory regions in secure world internal flash, and vice versa. The threat model does not allow the attacker to read arbitrary non-volatile memory bypassing the MCU, e.g. using unprotected debug features or via chip probing techniques. However, the software solution does not provide a checkpoint utility to restore the MCU to the previous state and perform intermittent computations.

SECure Context Saving (SECCS), based on a hardware module, stores the CPU content on a target NVM while providing confidentiality and integrity [22]. In [7], a checkpointing policy to determine when to checkpoint the MCU state is used to minimize the application time to completion while guaranteeing security.

Contribution to State-of-the-Art. In our approach, TrustZone for Cortex-M is used to provide strong isolation for a small (TCB) that contains the secure checkpoint utility and its related data. This is an improvement over the SICP implementation in [10] and [9], which relies on a weaker isolation from the MPU. TrustZone isolation is exploited in a similar fashion in [3], but our work also provides a full checkpoint utility to perform intermittent computation. Differently from [3], we store and secure the checkpoint on an external FRAM memory, so that the lifetime of the system is not severely limited by the wear of the internal flash, allowing for real-world applications.

Compared to [10] and [9], we relax the attacker capabilities to prevent access to internal memory, avoiding the need for tamper-free memory, which is not available on the considered MCUs. In addition, we highlight the importance of considering the entire security chain of the device. For example, [9], [10] do not discussed how an attacker with full write and read access to the internal memory could be prevented from modifying the firmware and accessing the protected secrets, even in the presence of tamper-free memory. Defending against such powerful attackers would require secure boot services and tamper-resistant memory.

4. Objectives, Attacker, Architecture

In the following sections, we implement and evaluate our approach to secure ImC on the STM32U585AII6Q with TrustZone. An established approach for protecting critical control systems with TrustZone is to place critical logic in the secure world and implement availability support for this critical logic [23], [24], effectively removing the normal world from the TCB. Our approach follows this model and focuses on securely checkpointing and restoring the secure world, yet it remains extensible to provide ImC capabilities to the normal world. Below we establish security objectives and define attacker capabilities.

Security Objectives. Our approach aims at achieving the same security objectives as earlier work on SICP [10]. In summary, we aim at guaranteeing:

- *Information security*: integrity, authenticity, and confidentiality to be assured for checkpointed data.
- *Freshness*: in order to prevent replay attacks, only the last valid checkpoint can be restored.
- *Atomicity*: power loss during checkpoint creation cannot leave the system in an undefined state upon startup, since an attacker can exploit this.
- *Unclonability*: it is not possible to clone the device from checkpoint data.

Forward progress is currently not assured for a device under attack at this moment. Related work does suggest that TrustZone is in principle capable of guaranteeing notions of progress under system compromise [2].

Attacker Model. We aim to protect against a strong adversary who can exploit software vulnerabilities to execute arbitrary code in the non-secure OS and who is assumed to have physical access to the device to, e.g., trigger a power outage or tamper with peripheral storage elements of the device, including our FRAM peripheral. We assume that the MCU's secure boot features and TrustZone isolation sufficiently protect the secure world and prevent software manipulation or the extraction of cryptographic keys from the secure world. Advanced hardware attacks that require opening and manipulating the MCU package to access the internal memories, as well as side-channel attacks against the secure world, are out of scope. Protection against those attacks is subject of orthogonal research.

Hardware Platform. The MCU used in this work is the STM32U585AII6Q by STMicroelectronics, provided with the B-U585I-IOT02A Discovery kit. The microcontroller features an ARM Cortex-M33, 2 Mbytes of internal flash memory divided into two banks, and 786 Kbytes of SRAM. The security features of the MCU relevant to this work are the following:

- ARM TrustZone for Cortex-M.
- MPU for privilege-based memory isolation.
- Random Number Generator (RNG).
- AES module with AES-GCM, 256-bit key size.
- Secure AES module (SAES) for side channel protection and key wrapping features.
- Readout protection (RDP) configurable on 3 levels, disabling static board configuration (option bytes) and debug features, and fixing the boot entry point.

The main limitation of this hardware platform is the absence of internal FRAM memory. The only non-volatile internal memory available in the STM32U5 devices is flash: Each 8 kB memory page is rated for 10.000 erase cycles, with 256 kB for each bank that can reach 100.000. This severely limits the lifetime of the device, if a checkpoint per capacitor discharge is considered: With frequent checkpointing and for large workloads and checkpoint sizes, this lifetime could be reduced to days or weeks.

Due to this reason, this work utilizes an external FRAM memory to store checkpoints. The Infineon CY15B102QN was chosen, featuring a 40MHz SPI interface, 10^{15} endurance cycles, low power modes, and 2MB of storage, which can fit two copies of the entire MCU SRAM. We still utilize the internal flash to store freshness information, which induces limited wear on the processor: If a checkpoint is created every second the lifetime of the memory would still be about 52 years. The 8-pin PDIP memory chip was placed on a breadboard and connected via jumper cables to the Arduino pins of the B-U585I-IOT02A Discovery kit.

5. Implementation

This section presents the secure checkpoint utility for saving the state of the secure world and discusses its security properties.

5.1. Memory Layout

The implementation required some changes to the default memory layout. This was achieved by modifying the linker script of the secure software image. The customized layout is shown in Fig. 1.

A *FLASH_CKP* memory area was carved inside the region of internal flash memory attributed to secure world (secure flash) to store checkpoint nonces. The size of

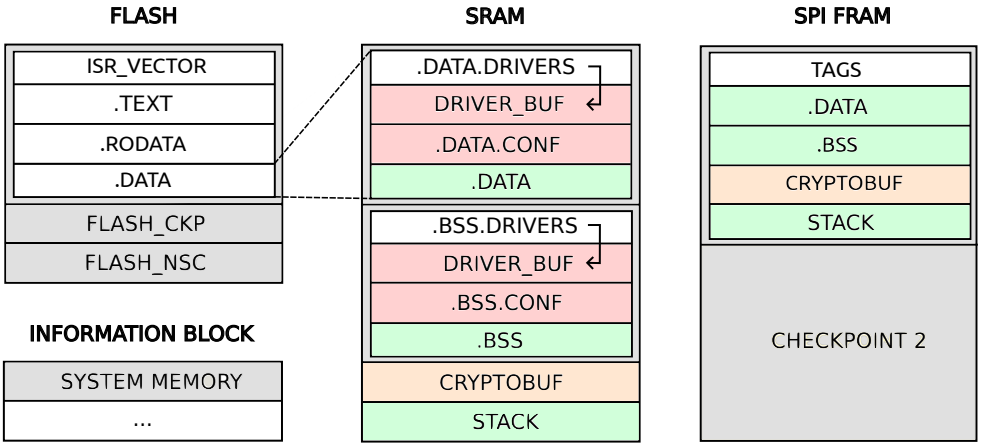

Figure 1. Memory layout of secure world and information block. Associated data (green), plaintext (red) and ciphertext (orange)

this area is 256 kB, since it is the maximum amount of flash memory that can sustain up to 100.000 erase cycles. The flash still contains the *.data* section, which stores initialized global variables (both global and static variables) and is copied inside the SRAM at first boot.

The structure of both the *.data* and the *.bss* sections, containing uninitialized global variables, was changed by adding the *.driver*, *driver_buffer*, and *.conf* sub-sections. The *.conf* sub-sections are used to store global variables that contain confidential information. The *.driver* sub-sections, instead, are designed to contain global variables that may change during the execution of the checkpoint utility (e.g. STM32 Hardware Abstraction Layer (HAL) drivers and checkpoint utility variables). These are copied inside the *driver_buffer* sub-sections before the start of the checkpoint creation procedure. The *.driver* sub-sections are also considered confidential. The *cryptobuf* section, instead, is a buffer to store the confidential information when encrypted.

The compile-time variable attribute *__attribute__((section("name")))* can be used to place the variables inside the *.conf* or *.driver* sub-sections. For third-party code, such as the STM32 HAL driver library, it is possible to place the variables from the entire object files inside the right sections via the linker script.

The external SPI memory is logically divided in half, to have space for two checkpoints. This is necessary to avoid corruption of the previous checkpoint in case of a failure during the creation of a new one.

Fig. 1 also shows the *information block*, which is a memory area inside bank 1 of the internal flash. The first 32 kB of the information block contain system memory. This area is immutable and reserved for use by STMicroelectronics. Among other things, it stores the 256-bit Root Hardware Unique Key (RHUK) of the device. This key is not accessible from software but it is directly wired to the SAES crypto unit. The RHUK is never used directly. It is used within TrustZone to generate Derived Hardware Unique Keys (DHUKs), which depend on the TrustZone state of the SAES module (secure or non-secure) and the key utilization mode (normal, wrapped, shared).

5.2. Checkpoint Utility

The checkpoint utility is composed of two functions: SAVE and RESTORE. SAVE is used to create and store new checkpoints in non-volatile memory, while RESTORE allows loading the saved state into SRAM when recovering from a power loss. The pseudocode for these functions is shown in Algorithm 1 and 2, respectively. The routines are invoked via system calls, since they need to run in privileged mode and need to be accessible from unprivileged software. Upon exit, the RESTORE utility jumps back to the application code, simulating an exit from the exception of the SAVE system call, which generated the restored checkpoint. The SAVE routine can be invoked periodically or just before the power loss, depending on the ImC system design.

SAVE. The first operation is to save data from the *.data.driver* and *.bss.driver* sub-sections inside the dedicated buffers in the new memory map (Algorithm 1, lines 1-2). This is required since the routine accesses the STM32 HAL driver library and some other global variables during execution, with the risk of checkpointing an inconsistent state. By placing their copy in the buffers, their state is preserved from possible changes during checkpoint operations.

The next operation is the generation of a random 96-bit nonce, by means of the Random Number Generator, to guarantee the freshness of the checkpoint (l. 3). The nonce will also act as the initialization vector (IV) of the authenticated encryption primitive, i.e., AES GCM.

Subsequently, the cryptographic primitive is called three times. The first call generates the authentication tag for the entire *.data* section, also encrypting the *driver_buffer* and the *.conf* sub-section, which are provided as the plaintext (**pt**), while the rest of the *.data* section is considered associated data (**ad**) and only authenticated (l. 4-6). The ciphertext (**ct**) is stored inside the *cryptobuf* (7). This partition is also shown in Fig. 1. The same operations are then performed for the *.bss* section (l. 8-11). The last call to the cryptographic primitive is used to generate the authentication tag for the stack (l. 12-13). At each invocation of AES GCM, the initialization vector is increased, preserving the security properties of the algorithm.

In the next step, tags, non-confidential *.data* and *.bss* sub-sections, *cryptobuf*, and the stack are saved in the external memory, at the currently selected checkpoint location (**ckp**) (l. 14-15). After this operation, the original IV is written to secure flash (l. 16). Since the internal flash memory requires the erasure of the entire 8-kB page to be rewritten, each IV is saved after the previous one. The minimum write size for the flash memory is 128 bits (quadword); since the IV is only 96 bits, the last word is used to store the address of the checkpoint location in the external memory (**ckp**). Only the last nonce stored in memory is considered valid. For this reason, when the new nonce is written to flash it invalidates the previous checkpoint. The MCU vendor does not specify whether flash writes are atomic, so it must be assumed that a power loss may result in a corrupted nonce. However, this does not introduce a security vulnerability because all checkpoints would be invalidated. Nonetheless, this could impede forward progress during normal operations.

It is also important to note that the new checkpoint is visible to the attacker before the invalidation of the previous one. If a counter was used as IV, instead of an RNG value, the attacker could remove power before the new nonce is saved to collect different AES-GCM messages generated with the same IV. This would introduce a security vulnerability.

The previous page is erased if the nonce is written at the beginning of a new flash page (l. 17-19). This allows the creation of a circular buffer for storing nonces. If a power failure happens during this operation, the page erasure is performed in the RESTORE routine. At this point, the storage locations for the next nonce and checkpoint are updated (l. 20-21), and main program execution resumes.

RESTORE. The first operation of the routine is loading the secret key inside the AES module (Algorithm 2, line 1). This should also be performed at the first system boot, since the key is also required by the SAVE routine. The secret key is not saved in plain text in the software image but is encrypted with the DHUK. The shared key mode is used to decrypt the secret key with the DHUK and to load it in the AES module (sharing).

The next step is to retrieve the latest nonce from secure flash, and the related checkpoint location (l. 2). This is done by reading the first 128 bits of each flash page to identify which one is currently used. A binary search is run on the page to find the latest nonce that was saved.

Non-confidential *.data* and *.bss* sub-sections, and the *cryptobuf* are restored from the external memory (l. 3). The *cryptobuf* is decrypted inside the confidential sub-sections and driver buffers of the *.data* (l. 4-7) and *.bss* (l. 8-11) sections. Authentication tags for the entire *.data* and *.bss* sections are also computed. The stack is restored from external memory and its authentication tag is generated (l. 12-14). All authentication tags are compared with the ones retrieved from the external memory (l. 15-18). Data inside the driver buffers is then copied to its original location (l. 19-20). When the restored nonce is the first one in a new flash page, the routine checks if the previous page was erased; otherwise, it is erased now (l. 21-23). Then the storage locations for the next nonce and checkpoint are set up (l. 24-25).

Algorithm 1 SAVE

1: $driverbuf_{.data} \leftarrow .data.drivers$
2: $driverbuf_{.bss} \leftarrow .bss.drivers$
3: $IV \leftarrow generateIV()$
4: $ad \leftarrow .data$
5: $pt \leftarrow driverbuf_{.data}|.data.conf$
6: $tag_1, ct \leftarrow AES\_GCM(IV, ad, pt)$
7: $cryptobuf.append(ct)$
8: $ad \leftarrow .bss$
9: $pt \leftarrow driverbuf_{.bss}|.bss.conf$
10: $tag_2, ct \leftarrow AES\_GCM(IV + 1, ad, pt)$
11: $cryptobuf.append(ct)$
12: $ad \leftarrow stack$
13: $tag_3 \leftarrow AES\_GCM(IV + 2, ad)$
14: $tags \leftarrow tag_1|tag_2|tag_3$
15: $writeSPI(tags|.data|.bss|cryptobuf|stack, ckp)$
16: $writeFlash(IV, ckp)$
17: **if** $firstNonceInNewPage()$ **then**
18: $\quad erasePreviousPage()$
19: **end if**
20: $ckp.toggleLocation()$
21: $flash.setNextNonceSlot()$

Algorithm 2 RESTORE

1: $unwrapSecretKey(key)$
2: $IV, ckp \leftarrow readFlash()$
3: $.data, .bss, cryptobuf \leftarrow readSPI()$
4: $ad \leftarrow .data$
5: $ct \leftarrow cryptobuf.at(driverbuf_{.data}|.data.conf)$
6: $tag_1, pt \leftarrow AES\_GCM^{-1}(IV, ad, ct)$
7: $driverbuf_{.data}|.data.conf \leftarrow pt$
8: $ad \leftarrow .bss$
9: $ct \leftarrow cryptobuf.at(driverbuf_{.bss}|.bss.conf)$
10: $tag_2, pt \leftarrow AES\_GCM^{-1}(IV, ad, ct)$
11: $driverbuf_{.bss}|.bss.conf \leftarrow pt$
12: $stack \leftarrow readSPI()$
13: $ad \leftarrow stack$
14: $tag_3 \leftarrow AES\_GCM^{-1}(IV, stack)$
15: $tags \leftarrow readSPI()$
16: **if** $tags! = tag_1|tag_2|tag_3$ **then**
17: $\quad abort()$
18: **end if**
19: $.data.drivers \leftarrow driverbuf_{.data}$
20: $.bss.drivers \leftarrow driverbuf_{.bss}$
21: **if** $firstNonceInNewPage()$ **then**
22: $\quad erasePreviousPage()$
23: **end if**
24: $ckp.toggleLocation()$
25: $flash.setNextNonceSlot()$

5.3. Security Discussion

The security of the device is based on the security of the RHUK. Since this key is stored in the flash bank 1 (in system memory), no protection is given from attackers who utilize chip probing techniques to read the content of internal flash. By accessing the RHUK and the encrypted checkpoint key in the software image (also stored in internal flash), they could retrieve the secret key used to secure the checkpoints. However, this is mitigated by the high cost of the instrumentation required for such attacks.

The next step in the chain is to ensure the TrustZone isolation of the secure world. This is configured in two steps: via option bytes and by the secure software at boot. Option bytes are stored in the information block (flash bank 1) and can be modified with debug access. The software images can also be read and modified if debug access is available. Therefore it is necessary to disable the debug port by setting the Readout Protection Level (RDP), which is also stored in the option bytes. With RDP=2 debug access is no longer possible.

Once the TrustZone isolation is in place, it is not possible to access secure memory areas (in both flash and SRAM), from non-secure software. This means that the secure software image cannot be read or modified from non-secure software, ensuring authenticity and confidentiality of both the software instructions and the related data. Authenticity and confidentiality are also ensured for the whole SRAM content. In addition, the isolation prevents non-secure software from accessing the cryptographic keys used to secure the checkpoints. As an implementation detail, DHUK is used to protect the encrypted checkpointing key, which is stored in secure flash and copied in secure SRAM at boot.

The checkpoint is stored in the external SPI memory by the SAVE routine. As discussed, the secrets used by the routines are protected, and therefore authenticity and confidentiality are ensured for the checkpoint. For what concerns freshness and state continuity, we follow the SICP approach [10]. The value of the nonce cannot be tampered with, because it is stored in secure flash. The SPI interface connected to the external memory is assigned to the secure world and thus inaccessible to normal-world software adversaries. Additional protection of software images can be provided via secure boot.

6. Benchmarks & Evaluation

Our checkpoint utility is evaluated in terms of the lifetime of the device and computational overhead.

6.1. Device Lifetime

The goal of intermittent computing systems is to enable a *deploy and forget* operational paradigm. Therefore, the lifetime of the device plays an essential role. The design choice to achieve this objective is to store checkpoints in an external FRAM memory, which, differently from internal flash, is not restricted regarding a limited amount of write cycles. However, internal flash is still required to store nonces. Considering a checkpoint area of 256 kB, which is the maximum amount of memory achieving 100,000 write cycles, 128 bit nonces and one checkpoint per second, the device lifetime becomes about 52 years.

6.2. Performance

The objective of the performance evaluation is to understand which is the overhead of the SAVE and RESTORE routines, which subtract energy from the main application. Two different evaluations are carried out. The first one tries to estimate the overall overhead of the SAVE and RESTORE routines when increasing the memory size of the application. The second one, instead, focuses on the overhead of the specific operations inside the routines. The chosen unit of measure is clock cycles, which can be converted to milliseconds given the 160 MHz clock frequency of the MCU.

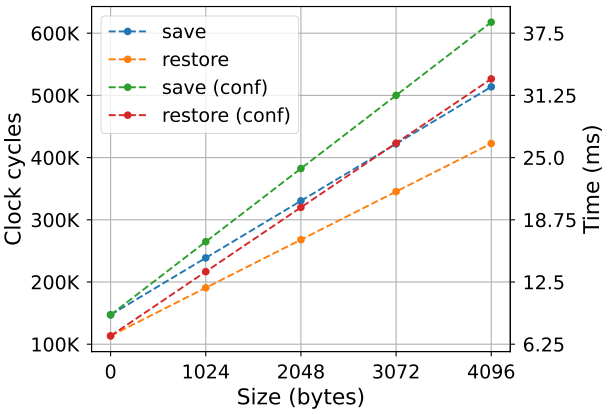

Figure 2. Comparison of SAVE and RESTORE overhead for varying application memory size

TABLE 1. BENCHMARK OF SAVE

Operation	Cycles	Time (ms)
Authenticated encryption	51777 (103745)	3,236 (6,484)
Copy drivers	11366	0,710
Write nonce	20409 (20408)	1,276
IV generation	451	0,028
Page erase*	274	0,017
SPI R/W	245883 (245884)	15,368
Others	275 (277)	0,017
Total	330435 (382405)	20,652 (23,9)

The test runs the external memory at 20 MHz, since it was the maximum stable frequency with the used setup. The DWT cycle counter is read before and after the monitored operation, and a message is sent via the UART interface containing the difference. For the evaluation of the total times, the DWT counter is only read before exception entry and after exception exit, to avoid additional overhead. When measuring single contributions, the code was changed to add DWT reads and UART communication inside the routines. Every measurement was repeated a minimum of ten times and then averaged.

Fig. 2 presents the results of the first evaluation. The x-axis shows the size of the main application's memory, which was simulated by defining an array of increasing size inside the *.bss* section. A null memory size indicates the overhead for saving the state of the checkpoint utility itself. Two opposite cases were considered: when the application memory is only authenticated and when it requires full confidentiality. Overhead increases linearly with application size for both SAVE and RESTORE operations. In addition, providing data confidentiality results in additional overhead w.r.t. only authentication. This is because, according to the AES GCM algorithm design, not encrypting the data saves processing time. SAVE also appears to be slower than RESTORE.

Tables 1 and 2 show the overhead for the single operations for an application size of 2 kB. The values

TABLE 2. BENCHMARK OF RESTORE

Operation	Cycles	Time (ms)
Authenticated encryption	52077 (104044)	3,255 (6,503)
Copy drivers	11325	0,708
Decrypt key	2697	0,169
Read nonce	586	0,037
Page erase*	274	0,017
SPI R/W	200677 (200675)	12,542
Others	378 (334)	0,024 (0,021)
Total	268014 (319935)	16,751 (19,996)

inside round brackets highlight differences when all the application memory is considered confidential, while the ones outside refer to authentication only. Since flash page erase is performed only when it is completely filled, its overhead was divided by the number of nonces that fit in a page, i.e., 512. It is possible to see that the biggest contributions for both SAVE and RESTORE are given by authenticated encryption and external memory operations. Also, the copy of the driver sub-sections results in significant overhead, but this is probably due to poor code optimization. The biggest differences between SAVE and RESTORE regard external memory operations and the programming of the nonce value in secure flash. Concerning the effect of confidentiality, the only significant difference is given by the cryptographic primitive.

Interesting observations can be made by comparing these results with the performance evaluation conducted for SICP [9]. When providing no security, the MSP430 requires less than one millisecond to both save and restore checkpoints with sizes ranging from 468 to 3198 bytes. This is a consequence of using internal FRAM: variables do not need to be copied since they already reside in non-volatile memory. The small overhead is mostly due to saving the peripheral state, a feature supported by TI's CTPL utility.

However, when authentication is introduced, the overhead is in the 20-73 ms range. These values are similar to what is obtained in our evaluation. Also, in this case, the MSP430 does not have to perform a memory copy, but only computes and saves the authentication tag. Therefore, by subtracting the SPI R/W contribution from our results, the STM32 hardware shows better performance. Finally, full confidentiality raises the SICP overhead to 68-380 ms. This is a consequence of additional cryptographic operations, which also require saving the ciphertext and the restored plaintext to memory.

In [1], checkpoint approaches supporting non-contiguous memory allocations and big program sizes with sparse memory access are tested for a Fast Fourier Transform (FFT) application. Results achieve a 50 to 150 ms overhead for checkpoint creation. Even if these approaches target additional functionalities w.r.t. our solution, they achieve comparable overheads without considering security.

It is important to note, however, that time is not the best metric for this type of comparison. ImC systems are constrained in terms of energy, so the best solution is not the fastest but the most efficient. The SM32U5 MCU was configured with the highest clock frequency of 160 MHz against the 8 MHz of the MSP430 processor in [9], likely leading to higher power consumption and wasting clock cycles while waiting for the completion of SPI and cryptographic operations. Future work should focus on analyzing and improving the energy efficiency of the proposed solution.

7. Conclusion & Future Work

Starting from the SICP protocol [9], [10], implemented on MSP430FR MCUs, we developed a secure checkpointing solution on the ARM Cortex-M platform. TrustZone for Cortex-M is used to provide isolation between a non-secure world, containing potentially vulnerable application software, and a secure world, containing secure trusted services, among which the checkpoint utility itself. Our implementation uses peripheral SPI FRAM memory to store checkpoints so as to protect the MCU from wear effects on the internal flash. Our utility relies on TrustZone's hardware keys to ensure integrity, confidentiality, and state continuity of system state in FRAM. We still rely on a small portion of the internal flash, in secure world memory, to store nonces that allow us to verify freshness and continuity, with very limited impact on the device's lifespan. Benchmarks are run to measure the computational overhead of the utility, establishing that the most demanding operations are the communications with the external memory and the cryptographic operations. We expect that our solution can be integrated with approaches that ensure real-time responsiveness of TrustZone applications under attack to develop notions of dependability for intermittent devices, securing emergent smart infrastructures, e.g., in smart agriculture or in the energy transition. Our prototype implementation for checkpointing and securely storing intermittent state on TrustZone-M is provided open-source under an MIT license at https://github.com/ptrchv/STM32-IntermittentSecurity.

This work faces some limitations that are subject to future research and implementation efforts. A crucial limitation of our prototype is that confidentiality of the checkpointed secure-world stack is not provided. However, our implementation provides authenticity and state continuity for the entire checkpoint, including the stack. While it is relatively easy to implement confidentiality protection for bounded stacks, handling dynamicity in workloads and stack utilization is difficult and requires additional implementation effort (cf. Appenix A). Furthermore, the state of peripherals and the state of non-secure world are not preserved. In immediate follow-up work, we will further benchmark our approach and develop a use case where measuring power consumption and effectiveness of our solution would be feasible.

Acknowledgements

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU. This research is partially funded by the CyberExcellence programme of the Walloon Region, Belgium (convention no. 2110186).

References

[1] S. Ahmed, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola, "Fast and Energy-Efficient State Checkpointing

for Intermittent Computing," *ACM Transactions on Embedded Computing Systems*, vol. 19, no. 6, pp. 45:1–45:27, Sep. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3391903

[2] F. Alder, G. Scopelliti, J. Van Bulck, and J. T. Mühlberg, "About time: On the challenges of temporal guarantees in untrusted environments," in *Workshop on System Software for Trusted Execution (SysTEX 2023)*, 2023.

[3] H. A. Asad, E. H. Wouters, N. A. Bhatti, L. Mottola, and T. Voigt, "On securing persistent state in intermittent computing," in *Proceedings of the 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, ser. ENSsys '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 8–14. [Online]. Available: https://doi.org/10.1145/3417308.3430267

[4] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini, "Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 1968–1980, 2016, conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. [Online]. Available: https://ieeexplore.ieee.org/document/7442814

[5] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini, "Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 15–18, Mar. 2015, conference Name: IEEE Embedded Systems Letters. [Online]. Available: https://ieeexplore.ieee.org/document/6960060

[6] N. A. Bhatti and L. Mottola, "HarvOS: Efficient Code Instrumentation for Transiently-Powered Embedded Sensing," in *2017 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, Apr. 2017, pp. 209–220. [Online]. Available: https://ieeexplore.ieee.org/document/7944791

[7] Z. Ghodsi, S. Garg, and R. Karri, "Optimal checkpointing for secure intermittently-powered IoT devices," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2017, pp. 376–383, iSSN: 1558-2434. [Online]. Available: https://ieeexplore.ieee.org/document/8203802

[8] H. Jayakumar, A. Raha, and V. Raghunathan, "QUICKRECALL: A Low Overhead HW/SW Approach for Enabling Computations across Power Cycles in Transiently Powered Computers," in *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, Jan. 2014, pp. 330–335, iSSN: 2380-6923. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6733152

[9] A. S. Krishnan and P. Schaumont, "Benchmarking and configuring security levels in intermittent computing," *ACM Trans. Embed. Comput. Syst.*, vol. 21, no. 4, sep 2022. [Online]. Available: https://doi.org/10.1145/3522748

[10] A. S. Krishnan, C. Suslowicz, D. Dinu, and P. Schaumont, "Secure intermittent computing protocol: Protecting state across power loss," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 734–739.

[11] A. Ltd, "Armv8-m memory model and memory protection user guide," https://developer.arm.com/documentation/107565/latest/.

[12] ——, "TrustZone for Cortex-M – Arm®," https://www.arm.com/technologies/trustzone-for-cortex-m.

[13] ——, "Trustzone technology for armv8-m architecture," https://developer.arm.com/documentation/100690/latest.

[14] ——, "Trustzone technology microcontroller system hardware design concepts user guide," https://developer.arm.com/documentation/107779/latest.

[15] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling, "Sancus 2.0: A Low-Cost Security Architecture for IoT Devices," *ACM Transactions on Privacy and Security*, vol. 20, no. 3, pp. 7:1–7:33, Jul. 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/3079763

[16] B. Ransford, J. Sorber, and K. Fu, "Mementos: system support for long-running computation on RFID-scale devices," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XVI. New York, NY, USA: Association for Computing Machinery, Mar. 2011, pp. 159–170. [Online]. Available: https://dl.acm.org/doi/10.1145/1950365.1950386

[17] A. S. Krishnan and P. Schaumont, "Exploiting security vulnerabilities in intermittent computing," in *Security, Privacy, and Applied Cryptography Engineering*, A. Chattopadhyay, C. Rebeiro, and Y. Yarom, Eds. Cham: Springer International Publishing, 2018, pp. 104–124.

[18] M. Schneider, R. J. Masti, S. Shinde, S. Capkun, and R. Perez, "Sok: Hardware-supported trusted execution environments," *arXiv preprint arXiv:2205.12742*, 2022.

[19] Texas Instruments, "TIDM-FRAM-CTPL reference design | TI.com," https://www.ti.com/tool/TIDM-FRAM-CTPL.

[20] Trusted Firmware, "TrustedFirmware-M (TF-M)," https://www.trustedfirmware.org/projects/tf-m/.

[21] S. Umesh and S. Mittal, "A survey of techniques for intermittent computing," *Journal of Systems Architecture*, vol. 112, p. 101859, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762120301430

[22] E. Valea, M. Da Silva, G. Di Natale, M.-L. Flottes, S. Dupuis, and B. Rouzeyre, "SI ECCS: SECure context saving for IoT devices," in *2018 13th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS)*, Apr. 2018, pp. 1–2. [Online]. Available: https://ieeexplore.ieee.org/document/8368561

[23] T. Van Eyck, H. Trimech, S. Michiels, D. Hughes, M. Salehi, H. Janjuaa, and T.-L. Ta, "Mr-tee: Practical trusted execution of mixed-criticality code," in *Proceedings of the 24th International Middleware Conference: Industrial Track*, ser. Middleware '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 22–28. [Online]. Available: https://doi.org/10.1145/3626562.3626831

[24] J. Wang, A. Li, H. Li, C. Lu, and N. Zhang, "Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 352–369.

[25] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2016.

[26] K. S. Yıldırım, "The today and future of intermittent computing for sustainable sensing," https://community.arm.com/arm-research/m/resources/990.

A. Gory Details

A.1. Limitations

A limitation of the current implementation is that the stack is not encrypted but only authenticated. This limitation can become a burden for the programmer, who currently should ensure that no confidential information is stored inside the stack, breaking the expected confidentiality guarantees of TrustZone. The reason for this limitation is that, due to the varying size of the stack, it is not possible to define a size for the *cryptobuf* to fully contain and encrypt it in one efficient operation. In addition, the HAL blocking driver for the cryptographic unit does not allow encryption of the same message in multiple calls. Therefore, encryption of the stack in different fixed-size portions would require the generation of multiple authentication tags, increasing the overall complexity of the solution. We currently do provide authenticity guarantees for the checkpointed stack since this does not require the allocation of a buffer for the cipher text. Similar considerations can be made regarding dynamic memory allocations inside the heap. However, in this case, the fragmentation induced by memory deallocation may pose

additional challenges which can be overcome by using custom allocator functions, for example.

Another important aspect is that interrupts cannot be fully disabled during the checkpointing routines. The reason is that the drivers for the AES module and SPI interface use timeouts based on interrupts from the systick timer. This could end up with the SAVE and RESTORE routines being interrupted, leading to inconsistencies inside the checkpoint. This could be prevented by carefully configuring interrupt priorities, or by changing the behavior of the drivers.

For ease of development and testing, some functionalities are missing from the provided code, which can trivially be added: No error handling in case of SAVE or RESTORE failures is implemented and the flash pages containing the nonces are not erased when they are filled. Secure boot is also not configured.

A.2. Possible Optimizations

For ease of development and testing, the code developed for this work lacks some optimization. The main one is the possibility of using the Direct Memory Access (DMA) controller to transfer data between the SRAM, cryptographic unit, and SPI interface. This would allow a parallelization of the checkpoint operations, leading to a lower overhead. This should be addressed in further research, also considering the effects on power consumption. In this regard, the MCU frequency could also be reduced, to avoid wasted CPU clock cycles when waiting for the SPI interface and cryptographic unit.

A.3. ARM TrustZone for Cortex-M

TrustZone isolation is managed by different MCU components [14]. The Implementation Defined Attribution Unit (IDAU) and the Security Attribution Unit (SAU), contained in the Cortex-M core, are used to specify the security attribution (S, NS, or NSC) of address space regions. The IDAU configuration is vendor-defined, while the SAU can be programmed in secure world (usually at boot). In case of conflict, the stricter configuration prevails. Bus transactions from the core are marked with the security attribute of the address space region they access. IDAU and SAU also prevent non-secure code from accessing secure regions.

However, due to hardware design constraints, the same peripheral components are mapped twice in the address space (aliasing) to reside in both secure and non-secure regions according to default IDAU partitions. Peripherals can be configured (from secure world only) to accept only one type of transaction (secure or non-secure), therefore deciding their security attribution. For memory peripherals, such as FLASH and SRAM, some areas can be attributed to secure and others to non-secure. Gates positioned between the bus and the peripherals implement the security checks. The design of peripheral gates and configuration components are delegated to MCU vendors.

In STM32U5 microcontrollers, TrustZone-aware peripherals contain TrustZone configuration registers. The others rely on the Global TrustZone Controller (GTZC) for their configuration. The STM32U5 MCU also contains additional bus masters, such as DMA controllers, that have no visibility on the policies defined in the IDAU and SAU. This can be a problem if they can be configured from non-secure world to access secure world regions. Therefore, they can be exclusively assigned to secure or non-secure world. Subsequently, the security attribution of their transactions will be checked by the gates of the accessed peripherals.