Exploiting Security Vulnerabilities in Intermittent Computing

Archanaa S. Krishnan(✉) and Patrick Schaumont(✉)

Virginia Tech, Blacksburg, VA 24060, USA
{archanaa,schaum}@vt.edu

Abstract. Energy harvesters have enabled widespread utilization of ultra-low-power devices that operate solely based on the energy harvested from the environment. Due to the unpredictable nature of harvested energy, these devices experience frequent power outages. They resume execution after a power loss by utilizing intermittent computing techniques and non-volatile memory. In embedded devices, intermittent computing refers to a class of computing that stores a snapshot of the system and application state, as a checkpoint, in non-volatile memory, which is used to restore the system and application state in case of power loss. Although non-volatile memory provides tolerance against power failures, they introduce new vulnerabilities to the data stored in them. Sensitive data, stored in a checkpoint, is available to an attacker after a power loss, and the state-of-the-art intermittent computing techniques fail to consider the security of checkpoints. In this paper, we utilize the vulnerabilities introduced by the intermittent computing techniques to enable various implementation attacks. For this study, we focus on TI's Compute Through Power Loss utility as an example of the state-of-the-art intermittent computing solution. First, we analyze the security, or lack thereof, of checkpoints in the latest intermittent computing techniques. Then, we attack the checkpoints and locate sensitive data in non-volatile memory. Finally, we attack AES using this information to extract the secret key. To the best of our knowledge, this work presents the first systematic analysis of the seriousness of security threats present in the field of intermittent computing.

Keywords: Intermittent computing · Attacking checkpoints
Embedded system security · Non-volatile memory

1 Introduction

Energy harvesters generate electrical energy from ambient energy sources, such as solar [JM17], wind [HHI+17], vibration [YHP09], electromagnetic radiation [CLG17], and radio waves [GC16]. Recent advances in energy-harvesting technologies have provided energy autonomy to ultra-low-power embedded devices. Since the energy is harvested depending on the availability of ambient energy, the harvester does not harvest energy continuously. Based on the

A. Chattopadhyay et al. (Eds.): SPACE 2018, LNCS 11348, pp. 104–124, 2018.
https://doi.org/10.1007/978-3-030-05072-6_7

availability of energy, the device is powered on/off, leading to an intermittent operation.

Classical devices come equipped with volatile memory, such as SRAM [AKSP18] or DRAM [NNM+18], which loses its state on power loss. In recent years, there has been a vast influx of devices with write efficient non-volatile memory, such as FRAM [YCCC07] or MRAM [SVRR13]. Non-volatile memory retains its state even after a power loss and provides instant on/off capabilities to intermittent devices. A majority of these devices contain both volatile and non-volatile memory. Typically, volatile memory is used to store the system and application state as it is relatively faster than non-volatile memory. The system state includes the processor registers, such as the program counter, stack pointer, and other general purpose registers, and settings of all the peripherals in use. The application state includes the stack, heap and any developer defined variables that are needed to resume program execution. And non-volatile memory is used to store the code sections, which is non-rewritable data. In the event of a power loss, volatile memory loses its program state, wiping both the application and system state. Thus, it is difficult to implement long-running applications on intermittent devices with only non-volatile memory to ensure accurate program execution.

Intermittent computing was proposed as a cure-all for the loss of program state and to ensure forward progress of long-running applications. Instead of restarting the device, intermittent computing creates a checkpoint that can be used to restore the device when power is restored. A checkpoint contains all the application and system state information necessary to continue the long-running application. It involves two steps: *checkpoint generation* and *checkpoint restoration*. In the checkpoint generation process, all the necessary information is stored as a checkpoint in non-volatile memory. When the device is powered up again, after a power loss, instead of restarting the application, checkpoint restoration is initiated. In the checkpoint restoration process, the system and application state are restored using the most recently recorded checkpoint, ensuring that the application resumes execution. There is extensive research in the field of intermittent computing, which is discussed further in the paper, that focuses on efficient checkpointing techniques for intermittent devices.

The introduction of non-volatile memory to a device changes the system dynamics by manifesting new vulnerabilities. Although the purpose of non-volatile memory is to retain checkpointed data even after a power loss, the sensitive data present in a checkpoint is vulnerable to an attacker who has access to the device's non-volatile memory. The non-volatile memory may contain passwords, secret keys, and other sensitive information in the form of checkpoints, which are accessible to an attacker through a simple JTAG interface or advanced on-chip probing techniques [HNT+13,SSAQ02]. As a result, non-volatile memory must be secured to prevent unauthorized access to checkpoints.

Recent work in securing non-volatile memory guarantees confidentiality of stored data [MA18]. Sneak -path encryption (SPE) was proposed to secure non-volatile memory using a hardware intrinsic encryption algorithm [KKSK15]. It

exploits physical parameters inherent to a memory to encrypt the data stored in non-volatile memory. iNVM, another non-volatile data protection solution, encrypts main memory incrementally [CS11]. These techniques encrypt the non-volatile memory in its entirety and are designed primarily for classical computers with unlimited compute power. We are unaware of any lightweight non-volatile memory encryption technique that can be applied to an embedded system. Consequently, a majority of the intermittent computing solutions do not protect their checkpoints in non-volatile memory [Hic17, JRR14, RSF11]. As far as we know, the state-of-the-art research in the intermittent computing field does not provide a comprehensive analysis of the vulnerabilities enabled by its checkpoints.

In this paper, we focus on the security of checkpoints, particularly that of intermittent devices, when the device is powered off. We study existing intermittent computing solutions and identify the level of security provided in their design. For evaluation purposes, we choose Texas Instruments'(TI) Compute Through Power Loss (CTPL) utility as a representative of the state-of-the-art intermittent computing solutions [Tex17a]. We exploit the vulnerabilities of an unprotected intermittent system to enable different implementation attacks and extract the secret information. Although the exploits will be carried out on CTPL utility, they are generic and can be applied to any intermittent computing solution which stores its checkpoints in an insecure non-volatile memory.

Contribution: We make the following contributions in this paper:

- We are the first to analyze the security of intermittent computing techniques and to identify the vulnerabilities introduced by its checkpoints.
- We implement TI's CTPL utility and attack its checkpoints to locate the sensitive variables of Advanced Encryption Standard (AES) in non-volatile memory.
- We then attack a software implementation of AES using the information identified from unsecured checkpoints.

Outline: Section 2 gives a brief background on existing intermittent computing solutions and their properties, followed by a detailed description of CTPL utility in Sect. 3. Section 4 details our attacker model. Section 5 enumerates the vulnerabilities of an insecure intermittent system, with a focus on CTPL utility. Section 6 exploits these vulnerabilities to attack CTPL's checkpoints to locate sensitive information stored in non-volatile memory. Section 7 utilizes the unsecured checkpoints to attack AES and extract the secret key. We conclude in Sect. 8.

2 Background on Intermittent Computing and Its Security

Traditionally, to generate a checkpoint of a long-running application, the application is paused before the intermittent computing technique can create a checkpoint. The process of saving and restoring the device state consumes extra energy

Table 1. A comparison of the state-of-the-art intermittent computing techniques based on the properties of its checkpoints and the nature of the checkpoint generation calls, such as online checkpoint calls, checkpoint calls placed around idempotent sections of code that do not affect the device state after multiple executions, voltage-aware techniques that dynamically checkpoint based on the input voltage, energy-aware techniques that dynamically generate checkpoints depending on the availability of energy and checkpoint (CKP) security

Intermittent model	Properties						
	Online	Idempotency	Voltage aware	Energy aware	HW	SW	CKP security
DINO [LR15]	–	–	–	–	–	✓	None
Mementos [RSF11]	–	–	–	✓	–	✓	None
QuickRecall [JRR14]	✓	–	✓	–	✓	✓	None
Clank [Hic17]	–	✓	–	–	✓	–	None
Ratchet [WH16]	–	✓	–	–	–	✓	None
Hibernus [BWM+15]	✓	–	–	✓	✓	✓	None
CTPL [Tex17a]	✓	–	✓	–	–	✓	None
Ghodsi et al. [GGK17]	✓	–	–	–	–	✓	Confidentiality

and time over the regular execution of the application, which is treated as the checkpoint overhead. This overhead depends on several factors such as the influx of energy, power loss patterns, progress made by the application, checkpoint size, and frequency of checkpoint generation calls. The latest intermittent computing techniques strive to be efficient, by minimizing the checkpoint overhead in their design. Table 1 compares various state-of-the-art intermittent computing techniques based on their design properties.

In DINO [LR15], Lucia et al. developed a software solution to maintain the volatile and non-volatile data consistency using a task-based programming and task-atomic execution model of an intermittent device. Ransford et al. [RSF11] developed Mementos, a software checkpointing system, which can be used without any hardware modifications. Mementos is an energy-aware checkpointing technique because checkpoint calls are triggered online depending on the availability of energy. At compile time, energy checks are inserted at the control points of the software program. At runtime, these checks trigger the checkpoint call depending on the capacitor voltage.

QuickRecall [JRR14], another online checkpointing technique, is a lightweight in-situ scheme that utilizes FRAM as a unified memory. When FRAM is utilized as a unified memory, it acts as both the conventional RAM and ROM. Now, FRAM contains both the application state from RAM and non-writable code sections from ROM. In the event of power-loss, RAM data remains persistent in FRAM and upon power-up, the program resumes execution without having to restore it. The checkpoint generation call is triggered upon detecting a drop in

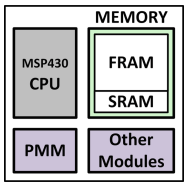

Fig. 1. MSP430FRxxxx architecture, contains the core (CPU), power management module (PMM), volatile memory (SRAM), non-volatile memory (FRAM) and other peripheral modules

Table 2. State of the core (CPU), the power management module (PMM), volatile memory (SRAM) and various clock sources (MCLK, ACLK, SMCLK) that drive various peripheral modules in different operating modes

Mode	CPU	PMM	SRAM	MCLK	ACLK	SMCLK
LPM0	On	On	On	On	On	Optional
LPM1	Off	On	On	Off	On	Optional
LPM2	Off	On	On	Off	On	Optional
LPM3	Off	On	On	Off	On	Off
LPM4	Off	On	On	Off	Off	Off
LPMx.5	Off	Off	Off	Off	Off	Off

the supply voltage. The net overhead incurred for checkpointing is reduced to storing and restoring the volatile registers that contain system state information. Apart from these energy-aware checkpointing techniques, other schemes have been proposed that leverages the natural idempotent properties of a program in their design [Hic17, WH16]. This property aids in identifying idempotent sections of code that can be executed multiple times and generate the same output every time.

None of the above intermittent computing solutions consider the security of its checkpoints, and the vulnerabilities introduced by non-volatile memory are ignored. An attacker with physical access to the device has the potential to read out the sensitive data stored in non-volatile memory. We know of one work which attempts to secure its checkpoints by encryption [GGK17]. Although encryption provides confidentiality, it does not guarantee other security properties, such as authenticity and integrity, without which an intermittent system is not fully secure because of the following reason. In all the latest checkpointing solutions, the device decrypts and restores the stored checkpoint without checking if it is a good or a corrupt checkpoint. If the attacker has the potential to corrupt the encrypted checkpoints, unbeknownst to the device, it will be restored to an attacker-controlled state. We exploit the lack of checkpoint security to mount our attacks in Sect. 7.

In the next section, we focus on TI's CTPL utility as an example of the latest intermittent computing solution.

3 CTPL

TI has introduced several low power microcontrollers in the MSP430 series. The FRAM series of devices, with a component identifier of the form MSP430FRxxxx, has up to 256 kB of on-chip FRAM for long-term data storage [Tex17b]. FRAM is an ideal choice of non-volatile memory for these microcontrollers for its high speed, low power, and endurance properties [KJJL05].

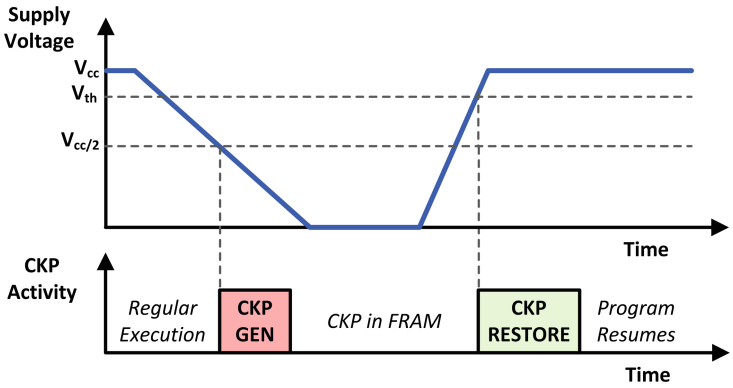

Fig. 2. Principle of operation of CTPL, checkpoint (CKP) generation and restoration based on the supply voltage, V_{cc}, and its set threshold voltage, V_{th}

Figure 1 illustrates the architecture of the FR series of devices. The MSP430 CPU is a 16-bit RISC processor with sixteen general purpose registers (GPR). The power management module (PMM) manages the power supply to CPU, SRAM, FRAM and other modules that are used. Typically, SRAM is the main memory that holds the application and system state. These microcontrollers can be operated in different operating modes ranging from Active Mode (AM) to various low power modes (LPM), listed in Table 2.

In active mode, PMM is enabled, which supplies the power supply to the device. The master clock (MCLK) is active and is used by the CPU. The auxiliary clock (ACLK), which is active, and subsystem master clock (SMCLK), which is either be active or disabled, are software selectable by the individual peripheral modules. For example, if a timer peripheral is used, it can either be sourced by ACLK or SMCLK, depending on the software program.

In low power modes, the microcontroller consumes lesser power compared to the active mode. The amount of power consumed in these modes depends on the type of LPM. Typically, there are five regular low power modes - LPM0 to LPM4; and two advanced low power modes - LPM3.5 and LPM4.5, also known as LPMx.5. As listed in Table 2, in all low power modes, the CPU is disabled as MCLK is not active. Apart from the CPU, other modules are disabled depending on its clock source. For instance, if a timer peripheral is sourced by SMCLK in active mode, this timer will be disabled in LPM3 as SMCLK is not active in this low power mode. But in all the regular low power modes, as PMM is enabled, SRAM remains active, which leaves the system and application state unchanged. Upon wakeup from a regular LPM, the device only needs to reinitialize the peripherals in use and continue with the retained SRAM state.

In LPMx.5, most of the modules are powered down, including PMM, to achieve the lowest power consumption of the device. Since PMM is no longer enabled, SRAM is disabled and the system and application state stored in SRAM are lost. Upon wakeup from LPMx.5, the core is completely reset. The applica-

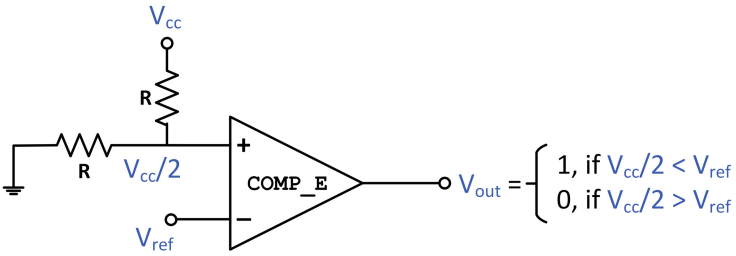

Fig. 3. Voltage monitor using comparator, `COMP_E`

tion has to reinitialize both the system and application state in SRAM, including the CPU state, required peripheral state, local variables, and global variables. Even though LPMx.5 is designed for ultra-low power consumption, the additional initialization requirement increases the start-up time and complexity of the application. TI introduced CTPL [Tex17a], a checkpointing utility that saves the necessary system and application state depending on the low power mode, to remove the dependency of saving and restoring state from the application.

CTPL utility also provides a checkpoint on-demand solution for intermittent systems, similar to QuickRecall [JRR14]. It defines dedicated linker description files for all its MSP430FRxxxx devices that allocates all the application data sections in FRAM and allocates a storage location to save volatile state information. Figure 2 illustrates the checkpoint generation and restoration process with respect to the supply voltage. A checkpoint is generated upon detecting power loss, which stores the volatile state information in non-volatile memory. Volatile state includes the stack, processor registers, general purpose registers and the state of the peripherals in use. Power loss is detected either using the on-chip analog-to-digital (ADC) converter or with the help of the internal comparator. Even after the device loses the main power supply, it is powered by the decoupling capacitors for a small time. The decoupling capacitors are connected to the power rails, and they provide the device with sufficient grace time to checkpoint the volatile state variables. After the required states are saved in a checkpoint, the device waits for a brownout reset to occur as a result of power loss. A timer is configured to timeout for false power loss cases when the voltage ramps up to the threshold voltage, V_{th}, illustrated in Fig. 2. Checkpoint restoration process is triggered by a timeout, device reset or power on, where the device returns to the last known application state using the stored checkpoint.

Using a Comparator to Detect Power Loss: The voltage monitor in Fig. 3 can be constructed using the comparator peripheral, `COMP_E`, in conjunction with an external voltage divider, to detect power loss. The input voltage supply, V_{CC}, is fed to an external voltage divider which provides an output equivalent to $V_{CC}/2$. The comparator is configured to trigger an interrupt if the output from the voltage divider falls below the 1.5 V reference voltage, V_{ref}, i.e, an interrupt is triggered if V_{CC} falls below 3 V. V_{ref} is generated by the on-chip reference

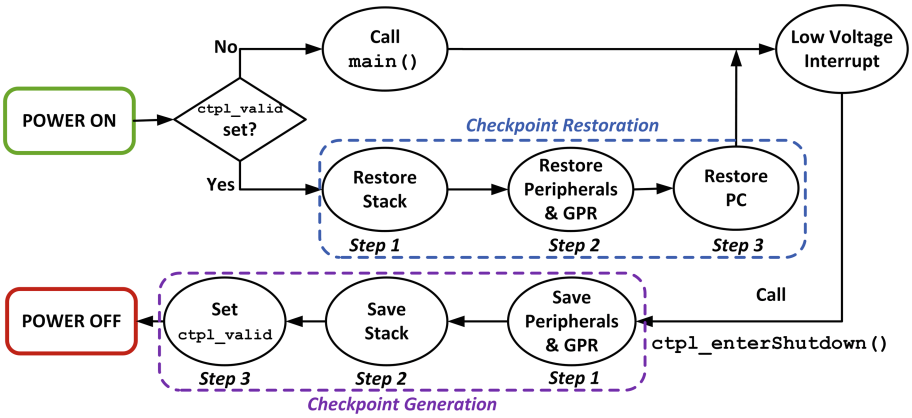

Fig. 4. CTPL checkpoint generation and restoration flowchart

module, `REF_A` [Tex17b]. The interrupt service routine will disable the voltage monitor and invoke the `ctpl_enterShutdown()` function, which saves the volatile state information.

Using ADC to Detect Power Loss: MSP430FRxxxx devices are equipped with a 12-bit ADC peripheral, `ADC12_B`, which can also be used to monitor the input voltage. Similar to the comparator based voltage monitor, the $V_{CC}/2$ signal is constantly compared to a fixed reference voltage to detect power loss. ADC peripheral is configured with the 2 V or 1.5 V reference voltage from the device's reference module, `REF_A`. $V_{CC}/2$ signal is provided by the internal battery monitor channel. The high side comparator is configured to 3.1 V. ADC monitor is triggered when the device has a stable input voltage of 3.1 V, upon which the device disables the high side trigger, enables the low side triggers, and begins monitoring V_{CC}. Upon detecting power loss the ADC monitor invokes `ctpl_enterShutdown()` function to save the volatile state information. The rest of the brownout and timeout functionalities are the same for the comparator and ADC based voltage monitor.

Checkpoint Generation: Call to `ctpl_enterShutdown()` function saves the volatile state in three steps, as shown in the bottom of Fig. 4. In the first step, the volatile peripheral state, such as a timer, comparator, ADC, UART, etc., and general purpose registers (GPRs) are stored in the non-volatile memory. The second and third step are programmed in assembly instructions to prevent mangling the stack when it is copied to the non-volatile memory. In the second step, the watchdog timer module is disabled to prevent unnecessary resets and the stack is saved. Finally, the `ctpl_valid` flag is set. `ctpl_valid` flag, which is a part of the checkpoint stored in FRAM, is used to indicate the completion of the checkpoint generation process and is set after the CTPL utility has checkpointed all the volatile state information. Until `ctpl_valid` is set, the system

does not have a complete checkpoint. After the flag is set, the device waits for a brownout reset or timeout. CTPL defines dedicated linker description files for all MSP430FRxxxx devices that places its application data sections in FRAM. Application specific variables, such as local and global variables, are retained in FRAM through power loss without explicitly storing or restoring them.

Checkpoint Restoration: Upon power-up, the start-up sequence checks if the `ctpl_valid` flag is set, as illustrated in Fig. 4. If the flag is set, then the non-volatile memory contains a valid checkpoint which can be used to restore the device, else the device starts execution from `main()`. Checkpoint restoration is also carried out in three steps. First, the stack is restored from the checkpoint location using assembly instructions, which resets the program stack. Second, CTPL restores the saved peripherals and general purpose registers before restoring the program counter in the final step. Then, the device jumps to the program counter set in the previous step and resumes execution.

In this complex mesh of checkpoint generation and restoration process of CTPL, checkpoint security is ignored. All the sensitive information from the application that is present in the stack, general purpose registers, local variables and global variables are vulnerable in the non-volatile memory. In the following sections, we describe our attacker model and enumerate various security risks involved in leaving checkpoints unsecured in a non-volatile memory.

4 Attacker Model

To evaluate the security of the current intermittent computing solutions, we focus on the vulnerabilities of the system when it is suspended after a power loss, and assume that the device incorporates integrity and memory protection features when it is powered on. We study two attack scenarios to demonstrate the seriousness of the security threats introduced by the checkpoints of an intermittent system. In the first case, we consider a *knowledgeable attacker* who has sufficient information about CTPL and the target device to attack the target algorithm. In the second case, we consider a *blind attacker* who does not have any information about CTPL or the target device but still possess the objective to attack the target algorithm. In both the cases, the attacker has the following capabilities.

- The attacker has physical access to the device.
- The attacker can access the memory via traditional memory readout ports or employ sophisticated on-chip probing techniques [HNT+13,SSAQ02], to retrieve persistent data. This allows unrestricted reads and writes to the data stored in the device memory, particularly the non-volatile memory, directly providing access to the checkpoints after a power loss. All MSP430 devices have a JTAG interface, which is mainly used for debugging and program development. We use it to access the device memory using development tools, such as TI's Code Composer Studio (CCS) and `mspdebug`.

– The attacker has sufficient knowledge about the target algorithm to analyze the memory. We assume that each variable of the target algorithm is stored in a contiguous memory location on the device. The feasibility of this assumption is described in Sect. 6 using Fig. 5
– The attacker can also modify the data stored in non-volatile memory without damaging the device. Therefore, the attacker has the ability to corrupt the checkpoints stored in non-volatile memory.

5 Security Vulnerabilities of Unsecured Checkpoints

Based on the above attacker model, we identify the following vulnerabilities, which are introduced by the checkpoints of an intermittent system.

Checkpoint Snooping: An attacker with access to the device's non-volatile memory has direct access to its checkpoints. Any sensitive data included in a checkpoint, such as secret keys, the intermediate state of a cryptographic primitive and other sensitive application variables, is now available to the attacker. Since CTPL is an open-source utility, a knowledgeable attacker can study the utility and easily identify the location of checkpoints, and in turn, extract sensitive information. A blind attacker can also extract sensitive information by detecting patterns that occur in memory. Section 6 provides a detailed description of techniques used in this paper to extract sensitive information. Vulnerable data, which is otherwise private during application execution, is now available for the attacker to use at their convenience. A majority of the intermittent computing techniques, similar to CTPL, do not protect their checkpoints. Although encrypting checkpoints protects the confidentiality of data, as in [GGK17], it is not sufficient to provide overall security to an intermittent system.

Checkpoint Spoofing: With the ability to modify non-volatile memory, the attacker can make unrestricted changes to checkpoints. In CTPL and other intermittent computing solutions, if a checkpoint exists, it is used to restore the device without checking if it is indeed an unmodified checkpoint of the current application setting. Upon power off, both the blind and knowledgeable attacker can locate the sensitive variable in a checkpoint, change it to an attacker-controlled value. As long as the attacker does not reset `ctpl_valid`, the checkpoint remains valid for CTPL. At the next power-up, unknowingly, the device restores this tampered checkpoint. From this point, the device continues execution in an attacker-controlled sequence. Encrypting checkpoints is not sufficient protection against checkpoint spoofing. The attacker can corrupt the encrypted checkpoint at random, and the device will decrypt and restore the corrupted checkpoint. Since the decrypted checkpoint may not necessarily correspond to a valid system or application state, the device may restore to an unstable state, leading to a system crash.

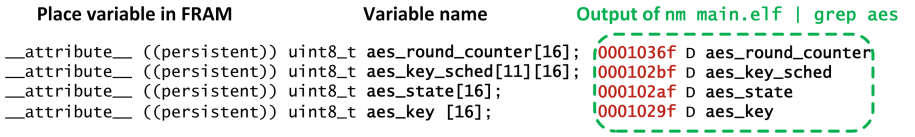

Fig. 5. AES variables present in a checkpoint and their contiguous placement in FRAM identified using the Linux command `nm`. `nm` lists the symbol value (hexadecimal address), symbol type (D for data section) and the symbol name present in the executable file `main.elf`.

Checkpoint Replay: An attacker who can snoop into the non-volatile memory can also make copies of all the checkpoints. Since both the blind and knowledgeable attackers are aware of the nature of the software application running on the device, they possess enough information to control the sequence of program execution. Equipped with the knowledge of the history of checkpoints, the attacker can overwrite the current checkpoint with any arbitrary checkpoint from their store of checkpoints. Since `ctpl_valid` is set in every checkpoint, the device is restored to a stale state from the replayed checkpoint. This gives the attacker capabilities to jump to any point in the software program with just a memory overwrite command. Similar to CTPL, the rest of the intermittent computing techniques also restore replayed checkpoints without checking if it is indeed the latest checkpoint.

6 Exploiting CTPL's Checkpoints

In this section, we provide a brief description of the software application under attack, followed by our experimental setup. We then explain our method to identify the location of checkpoints and sensitive data in FRAM, based on the capabilities of the attacker. We show that checkpoint snooping is sufficient to identify the sensitive data in non-volatile memory.

6.1 Experimental Setup

To mount our attack on CTPL utility, we used TI's MSP430FR5994 LaunchPad development board. The target device is equipped with 256 kB of FRAM which is used to store the checkpoints. We use TI's FRAM utility library to implement CTPL as a utility API [Tex17a]. We implement TI's software AES128 library on MSP430FR5994 as the target application running on the intermittent device. Figure 5 lists a minimum set of variables that must be checkpointed to ensure forward progress of AES. They are declared persistent to ensure that they are placed in FRAM. Figure 5 also lists the location of these variables in FRAM, identified using the Linux `nm` command. All the AES variables are placed next to each other in FRAM, from 0x1029F to 0x1037E, which satisfies our assumption that the variables of the target algorithm are stored in a contiguous memory

Fig. 6. Memory dump of FRAM, where the checkpoint begins from 0x10000 and ends at 0x103DB

location. The executable file,`main.elf`, was only used to prove the feasibility of this assumption and is not needed to carry out the attack described in this paper.

As CTPL is a voltage-aware checkpointing scheme, the application developer need not place checkpoint generation and restoration calls in the software program. CTPL, which is implemented as a library on top of the software program, automatically saves and restores the checkpoint based on the voltage monitor output. To access the checkpoints, we use `mspdebug` commands memory dump (`md`) and memory write (`mw`) to read from and write to the non-volatile memory, respectively, via the JTAG interface. Other memory probing techniques, [HNT+13,SSAQ02], can also be utilized to deploy our attack on AES when JTAG interface is disabled or unavailable.

6.2 Capabilities of a Knowledgeable Attacker

Armed with the information about CTPL and the target device, a knowledgeable attacker analyzes the 256 kB of FRAM to identify the location and size of checkpoints in non-volatile memory. The following analysis can be performed after CTPL generates at least one checkpoint, which is generated at random, on the target device.

Locate the Checkpoints in Memory: A knowledgeable attacker examines CTPL's linker description file for MSP430FR5994 to identify the exact location of FRAM region in the device's memory that hosts the checkpoints. In the linker description file, FRAM memory region is defined from 0x10000, which is the starting address of `.persistent` section of memory. CTPL places all application data sections in the `.persistent` section of the memory. Thus, the application specific variables required for forward progress are stored somewhere between in 0x10000 and 0x4E7FF.

Identifying Checkpoint Size: A knowledgeable attacker has the ability to distinguish the checkpoint storage from regular FRAM memory regions using two

```
diff md_001.txt md_010.txt
43,44c43,44
< :102A2:FF709AE3CC8FDB755E4FA44F2471AA09
< :102B2:3A0A68F305143A9F01A4CECE2084360 0
---
> :102A2:FF709AE3CC8FDB755E4FA44F2471AA1B
> :102B2:2A1B515A23F3B8C2995274B82AE25F00
```

```
diff md_001.txt md_008.txt
43,44c43,44
< :102A2:FF709AE3CC8FDB755E4FA44F2471AA09
< :102B2:3A0A68F305143A9F01A4CECE2084360 0
---
> :102A2:FF709AE3CC8FDB755E4FA44F2471AA64
> :102B2:57BDB25EE955979F01A4CECE2084360 0
```

```
diff md_003.txt md_006.txt
43,44c43,44
< :102A2:FF709AE3CC8FDB755E4FA44F2471AA7C
< :102B2:4F95825EE955979F01A4CECE2084360 0
---
> :102A2:FF709AE3CC8FDB755E4FA44F2471AA63
> :102B2:50B4BC5EE955979F01A4CECE2084360 0
```

```
diff md_001.txt md_007.txt
43,44c43,44
< :102A2:FF709AE3CC8FDB755E4FA44F2471AA09
< :102B2:3A0A68F305143A9F01A4CECE2084360 0
---
> :102A2:FF709AE3CC8FDB755E4FA44F2471AA68
> :102B2:5BA9AA5EE955979F01A4CECE2084360 0
```

Fig. 7. A section of the `diff` output of memory dumps that locates a consistent difference of 16 bytes at the memory location 0x102B0, which pinpoints the location of the intermediate state of AES

properties of the target device. First, any variable stored in FRAM must either be initialized by the program or it will be initialized to zero by default. Second, the target device's memory reset pattern is 0xFFFF. Based on these properties, the attacker determines that the checkpoint region of FRAM will either be initialized to a zero or non-zero value and the unused region of FRAM will retain the reset pattern. The knowledgeable attacker generates a memory dump of the entire FRAM memory region to distinguish the location of checkpoints. In the memory dump, only a small section of the 256 kB of FRAM was initialized, and the majority of the FRAM was filled with 0xFFFF, as shown in Fig. 6. Thus, the checkpoint is stored starting from 0x10000 up to 0x103DB, with a size of 987 bytes. In an application where the length of input and output are fixed, which is the case of our target application, the size of a checkpoint will remain constant. It is sufficient to observe this 987 bytes of memory to monitor the checkpoints.

Thus, a knowledgeable attacker who has access to the device's linker description file and device's properties can pinpoint the exact location of the checkpoint with a single copy of FRAM.

6.3 Capabilities of a Blind Attacker

Unlike knowledgeable attackers, blind attackers do not possess any information about CTPL or the device, but only have unrestricted access to the device memory. They can still analyze the device memory to locate sensitive information stored in it. The set capabilities of a knowledgeable attacker is a superset of the set of capabilities of a blind attacker. Therefore, the following analysis can also be performed by a knowledgeable attacker.

To ensure continuous operation of AES, CTPL stores the intermediate state of AES, `state`; secret key, `key`; round counter, `round` and other application variables in FRAM. These variables are present in every checkpoint and can be

identified by looking for a pattern in the memory after a checkpoint is generated. To study the composition of device memory, the blind attacker collects 100 different dumps of the entire memory of the device, where each memory dump is captured after a checkpoint is generated at a random point in AES, irrespective of the location and frequency of checkpoint calls. 100 was chosen as an arbitrary number of memory dumps to survey as a smaller number may not yield conclusive results. And a larger number will affirm the conclusions derived from 100 memory dumps. The blind attacker uses the following technique to locate `state` in the memory.

Locate the Intermediate State of AES: At a given point of time, AES operates on 16 bytes of intermediate state. This intermediate state is passed through 10 rounds of operation before a ciphertext is generated. By design, each round of AES confuses and diffuses its state such that at least half the state bytes are changed after every round. After two rounds of AES, all the 16 bytes of intermediate state are completely different from the initial state [DR02]. Thus, any 16 bytes of contiguous memory location that is different between memory dumps is a possible intermediate state. To identify the intermediate state accurately, the blind attacker stores each of the collected memory dump in an individual text file for post-processing using the Linux `diff` command. `diff` command locates the changes between two files by comparing them line by line. The attacker computes the difference between each of the 100 memory dumps using this command and makes the following observation. On average, seven differences appear between every memory dump. Six of the seven differences correspond to small changes to memory ranging from a single bit to a couple of bytes. Only one difference, located at 0x102A2, corresponds to a changing memory of up to 16 contiguous bytes, as shown in Fig. 7. Based on the design of AES, the attacker concludes that any difference in memory that lines up to a 16 bytes can be inferred as a change in `state`. From the `diff` output highlighted in Fig. 7, the blind attacker accurately identifies `state` to begin from 0x102B0 and end at 0x102BF. It is also reasonable to assume that `state` is stored in the same location in every checkpoint as it appears at 0x102B0 in all memory dumps.

The attacker can also pinpoint the location of the round counter using a similar technique. `round` is a 4-bit value that ranges from 0 to 11 depending on the different rounds of AES. Thus, any difference in memory that spans across 4 contiguous bits, and takes any value from 0 to 11 are ideal candidates for the round counter.

7 Attacking AES with Unsecured Checkpoints

Equipped with the above information on checkpoints and location of sensitive variables in FRAM, we extract the secret key using three different attacks - brute forcing the memory, injecting targeted faults in the memory and replaying checkpoints to enable side channel analysis. We demonstrate that when the attacker can control the location of checkpoint generation call, it is most efficient

to extract the secret key using fault injection techniques, and when the attacker has no control over the location of checkpoint call, brute forcing the key from memory yields the best results.

7.1 Brute Forcing the Key from Memory

Since the device must checkpoint all the necessary variables to ensure forward progress, it is forced to checkpoint the secret key used for encryption as well. To extract the key by brute forcing the memory, the attacker needs a checkpoint or a memory dump with a checkpoint, a valid plaintext/ciphertext pair, and AES programmed on an attacker-controlled device who's plaintext and key can be changed by the attacker. The attacker generates all possible keys from the memory, programs the attacker-controlled device with the correct plaintext and different key guesses. The key guess that generates the correct ciphertext output on the attacker-controlled device is the target device's secret key. Based on the assumption that the key stored in FRAM appears in 16 bytes of contiguous memory location, the attacker computes the number of possible keys using the following equation:

$$N_{KeyGuess} = L_{memory} - L_{key} + 1 \tag{1}$$

where, $N_{KeyGuess}$ is the total number of key guesses that can be derived from a memory, L_{memory} is the length of the memory in bytes and L_{key} is the length of key in bytes. The number of key guesses varies depending on the capabilities of the attacker, as detailed below.

Knowledgeable Attack: Knowledgeable attackers begins with a copy of a single checkpoint from FRAM. The 16-byte key is available in FRAM amidst the checkpointed data, which is 987 bytes long. Using Eq. 1, a knowledgeable attacker computes the number of possible key guesses to be 972. Thus, for a knowledgeable attacker, the key search space is reduced from 2^{128} to $2^9 + 460$.

Blind Attack: Since blind attackers do not know the location or size of the checkpoint, they start with a copy of the memory of the device that contains a single checkpoint. MSP430FR5994 has 256 kB of FRAM, which is 256,000 bytes long. Using Eq. 1, the number of key guesses for a blind attacker equals 255,985. For a blind attacker, the search space for the key is reduced to $2^{18} - 6159$

In both the attacker cases, all possible keys are derived by going over the memory 16 contiguous bytes at a time. These key guesses are fed to the attacker-controlled device to compute the ciphertext. The key guess that generates the correct ciphertext is found to be the secret key of AES. Even though a blind attacker generates more key guesses and requires more time, they can still derive the key in less than 2^{18} attempts, which is far less compared to the 2^{128} attempts of a regular brute force attack. The extracted key can be used to decrypt subsequent ciphertexts as long as it remains constant in checkpoints. If none of the

key guesses generate the correct ciphertext, then the secret was not checkpointed by CTPL. When the key is not stored in FRAM, it can be extracted using the two attacks described below.

7.2 Injecting Faults in AES via Checkpoints

Fault attacks alter the regular execution of the program such that the faulty behavior discloses information that is otherwise private. Several methods of fault injection have been studied by researchers, such as single bit faults [BBB+10] and single byte faults [ADM+10]. A majority of these methods require dedicated hardware support in the form of laser [ADM+10] or voltage glitcher [BBGH+14] to induce faults in the target device. Even with dedicated hardware, it is not always possible to predict the outcome of a fault injection. In this paper, we focus on injecting precise faults to AES and use existing fault analysis methods to retrieve the secret key.

To inject a fault on the target device, the attacker needs the exact location of the intermediate state in memory and the ability to read and modify the device memory. They also require a correct ciphertext output to analyze the effects of the injected fault. The correct ciphertext output is the value of `state` after the last round of AES, which is obtained from a memory dump of the device that contains a checkpoint that was generated after AES completed all ten rounds of operation. Both the blind and the knowledgeable attacker know the location of `state` in memory and have access to memory. A simple memory write command can change the state and introduce single or multiple bit faults in AES. This type of fault injection induces targeted faults in AES without dedicated hardware support. We describe our method to inject single bit and single byte fault to perform differential fault analysis (DFA) on AES introduced in [Gir05] and [DLV03] respectively.

Inducing Single Bit Faults: To implement the single-bit DFA described in [Gir05], the attacker requires a copy of the memory that contains a checkpoint that was generated just before the final round of AES. This memory contains the intermediate state which is the input to the final round. The attacker reads `state` from 0x102B0, modifies a single-bit at an arbitrary location in `state` and overwrites it with this faulty state to induce a single-bit fault. When the device is powered-up, CTPL restores the tampered checkpoint and AES resumes computation with the faulty state. The attacker then captures the faulty ciphertext output and analyzes it with the correct ciphertext to compute the last round key and subsequently the secret key of AES using the method described in [Gir05]. With the help of the unsecured checkpoints from CTPL, both blind and knowledgeable attackers can inject targeted faults in AES with single bit precision, enabling easy implementation of such powerful attacks.

Inducing Single Byte Faults: To induce a single byte fault and implement the attack described in [DLV03], the attacker requires a copy of the memory

that contains a checkpoint that was generated before the Mix Column transformation of the ninth round of AES. Similar to a single bit fault, the attacker overwrites `state` with a faulty state. The faulty state differs from the original `state` by a single byte. For example, if `state` contains 0x0F in the first byte, the attacker can induce a single byte fault by writing 0x00 to 0x102B0. When the device is powered-up again, CTPL restores the faulty checkpoint. AES resumes execution and the single byte fault is propagated across four bytes of the output at the end of the tenth round of AES. The faulty ciphertext differs from the correct ciphertext at memory locations 0x102B0, 0x102B7, 0x102BA and 0x102BD. Using this difference, the attacker derives all possible values for four bytes of the last round key. They induce other single byte faults in `state` and collect the faulty ciphertexts. They use the DFA technique described in [DLV03] to analyze the faulty ciphertext output and find the 16 bytes of AES key with less than 50 ciphertexts. Thus, the ability to modify checkpoints aids in precise fault injection which can be exploited to break the confidentiality of AES.

7.3 Replaying Checkpoints to Side Channel Analysis

The secret key of AES can also be extracted by using differential power analysis (DPA) [KJJ99]. In DPA, several power traces of AES are needed, where each power trace corresponds to the power required to process a different plaintext using the same secret key. These power traces are then analyzed to find the relation between the device's power consumption and secret bits, to derive the AES key.

Similar to DFA, to extract the secret key using DPA, the attacker needs the correct location of `state` of AES, which is known by both the blind and knowledgeable attacker. With access to the device memory, the attacker can read and modify `state` to enable DPA. To perform DPA on the target device, they need a copy of the device memory that contains a checkpoint that was generated just before AES begins computation. The `state` variable in this checkpoint contains the plaintext input to AES. It is sufficient to replay this checkpoint to restart AES computations multiple times. To obtain useful power traces from each computation, the attacker overwrites `state` with a different plaintext every time. Upon every power-up, CTPL restores the replayed checkpoint and AES begins computation with a different plaintext each time. The target device now encrypts each of the plaintext using the same key. The power consumption of each computation is recorded and processed to extract the secret bits leaked in the power traces, and consequently, derive the secret key. Even though this attack also requires a copy of memory and modifications to `state`, it requires other hardware, such as an oscilloscope, to collect and process the power traces to derive the secret key.

7.4 Attack Analysis

If it is feasible to obtain a copy of the memory that contains a checkpoint from a specified round of AES, then extracting the secret key by injecting faults in

checkpoints and performing DFA is the most efficient method for two reasons. First, DFA can extract secret key with less than 50 ciphertexts and an existing DFA technique, such as [DLV03,Gir05], but DPA requires thousands of power traces. Second, unlike DPA, DFA does not require hardware resources such as an oscilloscope to extract the secret key. Thus, injecting faults in checkpoints breaks the confidentiality of AES with the least amount of time and resources, compared to replaying checkpoints. If it not possible to determine when the checkpoint was generated, brute forcing the memory to extract the secret key is the only feasible option. All the attacks described in this paper can be carried out without any knowledge of the device or the intermittent computing technique in use. The attacker only needs unrestricted access to the non-volatile memory to extract sensitive data from it.

Apart from AES, the attacks explored in this paper are also effective against other cryptographic algorithms and security features, such as control flow integrity protection [DHP+15] and attestation solutions [EFPT12], that maybe implemented on an intermittent device. Thus, unprotected checkpoints undermine the security of the online protection schemes incorporated in intermittent devices.

8 Conclusions

Intermittent computing is emerging as a widespread computing technique for energy harvested devices. Even though several researchers have proposed efficient intermittent computing techniques, the security of such computing platforms is not a commonly explored problem. In this paper, we study the security trends in the state-of-the-art intermittent computing solutions and investigate the vulnerabilities of the checkpoints of CTPL. Using the unsecured checkpoints, we demonstrate several attacks on AES that was used to retrieve the secret key. This calls for intermittent computing designs that address the security pitfalls introduced in this paper. Since security is not free, resource constrained devices require lightweight protection schemes for their checkpoints. Hence, dedicated research is needed to provide comprehensive, energy efficient security to intermittent computing devices.

Acknowledgements. This work was supported in part by NSF grant 1704176 and SRC GRC Task 2712.019.

References

[ADM+10] Agoyan, M., Dutertre, J.M., Mirbaha, A.P., Naccache, D., Ribotta, A.L., Tria, A.: Single-bit DFA using multiple-byte laser fault injection. In: 2010 IEEE International Conference on Technologies for Homeland Security (HST), pp. 113–119, Novomber 2010

[AKSP18] Afzali-Kusha, H., Shafaei, A., Pedram, M.: A 125mV 2ns-access-time 16Kb SRAM design based on a 6T hybrid TFET-FinFET cell. In: 2018 19th International Symposium on Quality Electronic Design (ISQED), pp. 280–285, March 2018

[BBB+10] Barenghi, A., Bertoni, G.M., Breveglieri, L., Pellicioli, M., Pelosi, G.: Fault attack on AES with single-bit induced faults. In: 2010 Sixth International Conference on Information Assurance and Security, pp. 167–172, August 2010

[BBGH+14] Beringuier-Boher, N., et al.: Voltage glitch attacks on mixed-signal systems. In: 2014 17th Euromicro Conference on Digital System Design, pp. 379-386, August 2014

[BWM+15] Balsamo, D., Weddell, A.S., Merrett, G.V., Al-Hashimi, B.M., Brunelli, D., Benini, L.: Hibernus: sustaining computation during intermittent supply for energy-harvesting systems. IEEE Embed. Syst. Lett. **7**(1), 15–18 (2015)

[CLG17] Chaari, M.Z., Lahiani, M., Ghariani, H.: Energy harvesting from electromagnetic radiation emissions by compact flouresent lamp. In: 2017 Ninth International Conference on Advanced Computational Intelligence (ICACI), pp. 272–275, February 2017

[CS11] Chhabra, S., Solihin, Y.: i-NVMM: a secure non-volatile main memory system with incremental encryption. In: 38th International Symposium on Computer Architecture (ISCA 2011), San Jose, CA, USA, 4–8 June 2011, pp. 177–188 (2011)

[DHP+15] Davi, L., et al.: HAFIX: hardware-assisted flow integrity extension. In: 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6, June 2015

[DLV03] Dusart, P., Letourneux, G., Vivolo, O.: Differential fault analysis on A.E.S. In: Zhou, J., Yung, M., Han, Y. (eds.) ACNS 2003. LNCS, vol. 2846, pp. 293–306. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45203-4_23

[DR02] Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Springer, Heidelberg (2002). https://doi.org/10.1007/978-3-662-04722-4

[EFPT12] El Defrawy, K., Francillon, A., Perito, D., Tsudik, G.: SMART: secure and minimal architecture for (establishing a dynamic) root of trust. In: NDSS: 19th Annual Network and Distributed System Security Symposium, San Diego, USA, 5–8 February 2012 (2012)

[GC16] Ghosh, S., Chakrabarty, A.: Green energy harvesting from ambient RF radiation. In: 2016 International Conference on Microelectronics, Computing and Communications (MicroCom), pp. 1–4, January 2016

[GGK17] Ghodsi, Z., Garg, S., Karri, R.: Optimal checkpointing for secure intermittently-powered IoT devices. In: 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 376–383, November 2017

[Gir05] Giraud, C.: DFA on AES. In: Dobbertin, H., Rijmen, V., Sowa, A. (eds.) AES 2004. LNCS, vol. 3373, pp. 27–41. Springer, Heidelberg (2005). https://doi.org/10.1007/11506447_4

[HHI+17] Habibzadeh, M., Hassanalieragh, M., Ishikawa, A., Soyata, T., Sharma, G.: Hybrid solar-wind energy harvesting for embedded applications: supercapacitor-based system architectures and design tradeoffs. IEEE Circuits Syst. Mag. **17**(4), 29–63 (2017)

[Hic17] Hicks, M.: Clank: architectural support for intermittent computation. In: Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, pp. 228–240. ACM, New York (2017)

[HNT+13] Helfmeier, C., Nedospasov, D., Tarnovsky, C., Krissler, J.S., Boit, C., Seifert, J.-P.: Breaking and entering through the silicon. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & #38; Communications Security, CCS 2013, pp. 733–744. ACM, New York (2013)

[JM17] Jokic, P., Magno, M.: Powering smart wearable systems with flexible solar energy harvesting. In: 2017 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1–4, May 2017

[JRR14] Jayakumar, H., Raha, A., Raghunathan, V.: 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, pp. 330–335, January 2014

[KJJ99] Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48405-1_25

[KJJL05] Kim, K., Jeong, G., Jeong, H., Lee, S.: Emerging memory technologies. In: Proceedings of the IEEE 2005 Custom Integrated Circuits Conference, pp. 423–426, September 2005

[KKSK15] Kannan, S., Karimi, N., Sinanoglu, O., Karri, R.: Security vulnerabilities of emerging nonvolatile main memories and countermeasures. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **34**(1), 2–15 (2015)

[LR15] Lucia, B., Ransford, B.: A simpler, safer programming and execution model for intermittent systems. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pp. 575–585. ACM, New York (2015)

[MA18] Mittal, S., Alsalibi, A.I.: A survey of techniques for improving security of non-volatile memories. J. Hardw. Syst. Secur. **2**(2), 179–200 (2018)

[NNM+18] Navarro, C., et al.: InGaAs capacitor-less DRAM cells TCAD demonstration. IEEE J. Electron Dev. Soc. **6**, 884–892 (2018)

[RSF11] Ransford, B., Sorber, J., Kevin, F.: Mementos: system support for long-running computation on RFID-scale devices. SIGARCH Comput. Archit. News **39**(1), 159–170 (2011)

[SSAQ02] Samyde, D., Skorobogatov, S., Anderson, R., Quisquater, J.J.: On a new way to read data from memory. In: Proceedings of First International IEEE Security in Storage Workshop, pp. 65–69, December 2002

[SVRR13] Sharad, M., Venkatesan, R., Raghunathan, A., Roy, K.: Multi-level magnetic RAM using domain wall shift for energy-efficient, high-density caches. In: International Symposium on Low Power Electronics and Design (ISLPED), pp. 64–69, September 2013

[Tex17a] Texas Instruments: MSP MCU FRAM Utilities (2017)

[Tex17b] Texas Instruments: MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family User's Guide (2017)

[WH16] Van Der Woude, J., Hicks, M.: Intermittent computation without hardware support or programmer intervention. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016), pp. 17–32. USENIX Association, Savannah (2016)

[YCCC07] Yang, C.F., Chen, K.H., Chen, Y.C., Chang, T.C.: Fabrication of one-transistor-capacitor structure of nonvolatile TFT Ferroelectric RAM devices using BA(Zr0.1 Ti0.9)O3 gated oxide film. IEEE Trans. Ultrason. Ferroelectr. Freq. Control **54**(9), 1726–1730 (2007)

[YHP09] Yun, S.-N., Ham, Y.-B., Park, J.H.: Energy harvester using PZT actuator with a cantilver. In: 2009 ICCAS-SICE, pp. 5514–5517, August 2009