Top-down Approach To Securing Intermittent Embedded Systems

Archanaa Santhana Krishnan

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Engineering

September 2, 2021

Blacksburg, Virginia

Keywords: Security, Intermittent computing, Exploiting checkpoints, Benchmarks

Top-down Approach To Securing Intermittent Embedded Systems

Archanaa Santhana Krishnan

(ABSTRACT)

The conventional computing techniques are based on the assumption of a near constant source of input power. While this assumption is reasonable for high-end devices such as servers and mobile phones, it does not always hold in embedded devices. An increasing number of Internet of Things (IoTs) is powered by intermittent power supplies which harvest energy from ambient resources, such as vibrations. While the energy harvesters provide energy autonomy, they introduce uncertainty in input power. Intermittent computing techniques were proposed as a coping mechanism to ensure forward progress even with frequent power loss. They utilize non-volatile memory to store a snapshot of the system state as a checkpoint. The conventional security mechanisms do not always hold in intermittent computing. This research takes a top-down approach to design secure intermittent systems. To that end, we identify security threats, design a secure intermittent system, optimize its performance, and evaluate our design using embedded benchmarks. First, we identify vulnerabilities that arise from checkpoints and demonstrates potential attacks that exploit the same. Then, we identify the minimum security requirements for protecting intermittent computing and propose a generic protocol to satisfy the same. We then propose different security levels to configure checkpoint security based on application needs. We realize configurable intermittent security to optimize our generic secure intermittent computing protocol to reduce the overhead of introducing security to intermittent computing. Finally, we study the role of application in intermittent computing and study the various factors that affect the forward progress of applications in secure intermittent systems. This research highlights that power loss is a threat vector even in embedded devices, establishes the foundation for security in intermittent computing.

Top-down Approach To Securing Intermittent Embedded Systems

Archanaa Santhana Krishnan

(GENERAL AUDIENCE ABSTRACT)

The embedded systems are present in every aspect of life. They are available in watches, mobile phones, tablets, servers, health aids, home security, and other everyday useful technology. To meet the demand for powering up a rising number of embedded devices, energy harvesters emerged as a solution to provide an autonomous solution to power on low-power devices. With energy autonomy, came energy scarcity that introduced intermittent computing, where embedded systems operate intermittently because of lack of constant input power. The intermittent systems store snapshots of their progress as checkpoints in non-volatile memory and restore the checkpoints to resume progress. On the whole, the intermittent system is an emerging area of research that is being deployed in critical locations such as bridge health monitoring. This research is focused on securing intermittent systems comprehensively. We perform a top-down analysis to identify threats, mitigate them, optimize the mitigation techniques, and evaluate the implementation to arrive at secure intermittent systems. We identify security vulnerabilities that arise from checkpoints to demonstrate the weakness in intermittent systems. To mitigate the identified vulnerabilities, we propose secure intermittent solutions to protect intermittent systems using a generic protocol. Based on the implementation of the generic protocol and its performance, we propose several optimizations based on the needs of the application to securing intermittent systems. And finally, we benchmark the security properties using two-way relation between security and application in intermittent systems. With this research, we create a foundation for designing secure intermittent systems.

Dedicated to Bhama Paati

Acknowledgments

My PhD journey is enriched with assistance from many people.

First and foremost I would like to thank my advisor, Dr. Patrick Schaumont, for his advice during the course of my Ph.D. His patience, immense knowledge, and tutelage were instrumental in the completion of this dissertation. I would like to thank my committee members, Dr. Ha, Dr. Nazhand-ali, Dr. Gerdes, and Dr. Hicks, for their feedback which helped shape my experiments and results. I also thank my committee co-chair, Dr. Dong Ha, for his help in transitioning between Virginia Tech and WPI.

I would like to thank all the current and past members of the Secure Embedded Systems Lab whose help and support made my research and my life in Blacksburg and Worcester a pleasant journey. I would like to thank National Science Foundation (NSF) and Semiconductor Research Corporation (SRC) for the research assistantship that made it possible to conduct this research.

I would like to thank all my friends, from VIT, VT, and WPI, who have been through the thick and thin of my research journey with me with their unparalleled support and timely tomfoolery. I am grateful to them for listening to my research musing and random facts, a habit I have developed during my Ph.D. Shreya, my roommate turned friend, I am grateful to you for creating a home away from home for us. Many thanks to all the aunts, uncles, and cousins for their encouragement and support throughout my Ph.D.

I would like to thank Amma, Appa, and Ammu for their unconditional support. And to Putu-Putu, for giving me a smile when I needed one.

Contents

List of Figures

List of Tables

List of my Publications

[1] Archanaa S. Krishnan and Patrick Schaumont. Configuring And Benchmarking Security For Intermittent Computing. Under review.

[2] Xiang Cheng, Hanchao Yang, Archanaa S. Krishnan, Patrick Schaumont and Yaling Yang. KHOVID: Interoperable Privacy Preserving Digital Contact Tracing. December 2020 arXiv 2012.09375.

[3] Pantea Kiaei, Archanaa S. Krishnan, and Patrick Schaumont. Parallel Synchronous Code Generation for Second Round Light Weight Candidates. NIST 4th Workshop on Lightweight Cryptography, October 2020

[4] Archanaa S. Krishnan, Charles Suslowicz, and Patrick Schaumont. 2020. Secure and Stateful Power Transitions in Embedded Systems. Journal of Hardware and System Security, Special Issue on SPACE 2018.

[5] Archanaa S. Krishnan, and Patrick Schaumont. Risk and Architecture factors in Digital Exposure Notification. International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS XX), July 2020. doi. Extended version in IACR ePrint Archive 2020/582.

[6] Archanaa S. Krishnan, and Patrick Schaumont. Hardware Support for Secure Intermittent Architectures (Extended Abstract). Workshop on Energy-Secure System Architectures (ESSA), May 2019.

[7] Daniel Dinu, Archanaa S. Krishnan, and Patrick Schaumont. SIA: Secure Intermittent Architecture for Off-the-Shelf Resource-Constrained Microcontrollers.IEEE International

Symposium on Hardware Oriented Security and Trust (HOST 2019), McLean, VA, May 2019.

[8] Archanaa S. Krishnan Charles Suslowicz, Daniel Dinu, and Patrick Schaumont. Secure Intermittent Computing Protocol: Protecting State Across Power Loss. Design, Automation & Test in Europe (DATE 2019), Florence, Italy, March 2019.

[9] Archanaa S. Krishnan and Patrick Schaumont. Exploiting Security Vulnerabilities in Intermittent Computing. 8th International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE 2018), Kanpur, India, December 2018.

[10] Charles Suslowicz, Archanaa S. Krishnan, Daniel Dinu, and Patrick Schaumont.Secure Application Continuity in Intermittent Systems. 9th International Green and Sustainable Computing Conference (IGSCC 2018), Pittsburg, PA, October 2018.

[11] Charles Suslowicz, Archanaa S. Krishnan , and Patrick Schaumont. Optimizing Cryptography in Energy Harvesting Applications. In Proceedings of the 2017 Workshop on Attacks and Solutions in Hardware Security (ASHES '17). ACM, New York, NY, USA, 17-26.

Chapter 1

Introduction

The Internet of Things (IoT) is an evolving technology that fosters connectivity between devices. The IoT supports a virtual representation of the real world through sensors and actuators, and it enables significant opportunities for optimization and analysis in smart grids, smart homes, smart cities, smart hospitals, and others. The scale of the IoT is enormous. By 2025, the number of computing devices in the IoT is projected to increase to 75 billion, and the data volume from these devices will exceed 79 zettabytes [18]. The number of devices on the one hand, and the level of trust placed in them on the other hand, have important implications for the realization of IoT devices.

Typically, IoT devices were powered by the grid or by a battery, both of which are not optimal power supply solutions. The placement of IoT devices is restricted by the availability of grid power supply. To overcome this issue, small batteries were used to power certain embedded devices to wean off the dependency on grid power. Batteries have two disadvantages. First, the production and disposal of batteries are harmful to the environment including contamination of drinking water and land [46, 74]. Second, the batteries need to be replaced periodically, depending on the battery capacity, application, and embedded device [77]. There was a need for energy autonomous embedded devices that could operate without restrictions from the grid power supply and periodic battery replacement. This was answered by energy harvesters.

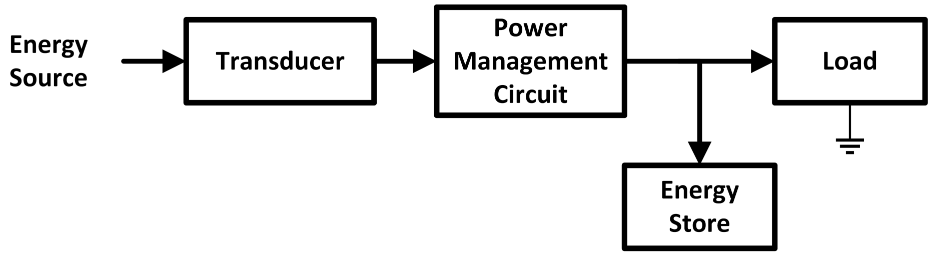


Figure 1.1: Three components of an energy harvester: (1) a transducer to convert ambient energy to electrical energy, (2) a power management circuit to adjust the harvested energy based on the needs of the load, and (3) an energy storage buffer to deliver high power to the load

1.1 Energy Harvester

Energy harvesters generate electrical energy from ambient energy sources, such as solar [45], wind [38], vibration [104], electromagnetic radiation [23], and radio waves [33]. The ambient energy is processed in three steps before it is consumed by the load, illustrated in Figure 1.1. First, a transducer converts ambient energy to electrical energy. Second, a power management circuit efficiently manages the harvested energy based on the requirements of the load. Since the harvester typically supplies low power, third, a supercapacitor or a battery is used as an energy storage buffer to accumulate the harvested electrical energy to supply bursts of high power to the load. Recent advances in energy-harvesting technologies have provided energy autonomy to low-power embedded devices [55, 80, 98].

Classical devices come equipped with volatile memory, such as SRAM [6] or DRAM [68], which loses its state on power loss. The ambient energy sources depend on external factors, including, but not limited to, weather, time of day, human activity, and location of the harvester. For example, sunlight is only available during the daytime and is dependent on the weather conditions; wind energy is similarly dependent on the weather; kinetic energy is dependent on the machine or human motion. Based on the availability of energy, the device is powered on/off, leading to an intermittent operation.

Table 1.1: Comparing the features of various memory technologies such as SRAM, Flash and EPROM with emerging non-volatile memory technologies including FRAM, MRAM, PCM, and NRAM [95]

	SRAM	**EPROM**	**Flash**	**Emerging NVM**
Non-volatility	No	Yes	Yes	Yes
Write performance	↑	↓	↓	↑
Read performance	↑	-	-	↑
Endurance	↑	↓	↓	↑
Power	↓	↑	↑	↓

↓: Low, ↑: High

1.2 Non-volatile memory

Non-volatile memory retains its state even after a power loss. Several commercially available embedded devices are equipped with Flash technology as non-volatile memory [1, 2]. While Flash memory was a key technology in bringing non-volatile memory to embedded devices, it has certain limitations that are not ideal for resource constrained devices. First, Flash has limited endurance which restricts the number of times Flash can be erased and re-programmed. Second, Flash has asymmetrical read and write overhead with power hungry and high latency writes. In recent years, there has been a vast influx of devices with write efficient non-volatile memory, such as Ferroelectric RAM(FRAM) [103], Phase change memory (PCM) [5], Magnetoresistive RAM, and (MRAM) [85]. While MRAM and NRAM technologies are only available as stand-alone memory, FRAM and PCM are available as embedded memory in commercial microcontroller lines from Texas Instruments [97] and STMicroelectronics [5] respectively. Table 1.1 compares various non-volatile memory technologies available to highlight the advantages of emerging non-volatile memory over traditional Flash and EPROM.

A majority of the latest embedded devices contain both volatile and non-volatile memory. Typically, volatile memory is used to store the system and application state as it is relatively

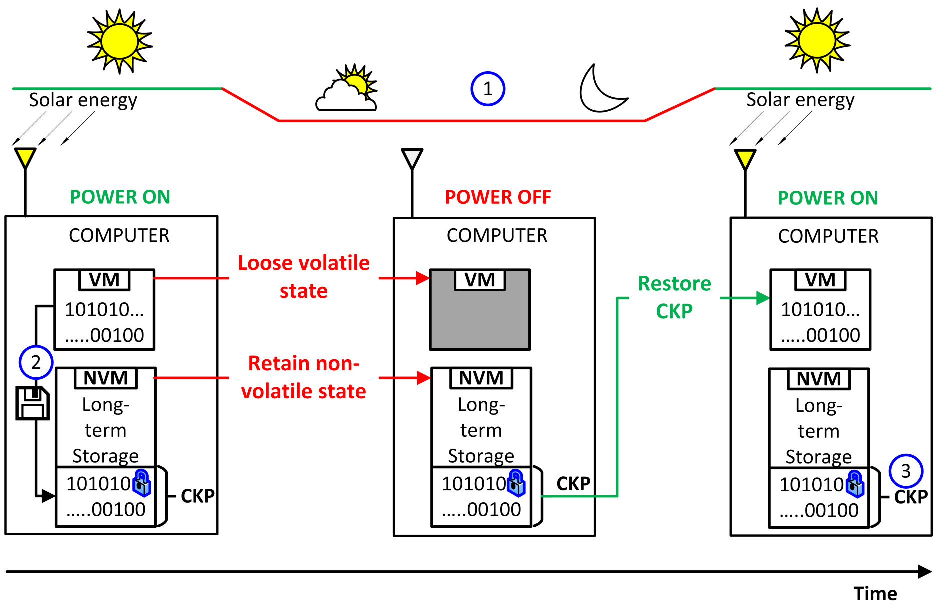


Figure 1.2: Flow of intermittent computing: (1) Cause - lack of solar energy causes power loss, (2) Statefulness - the state of volatile memory, VM, is stored as checkpoints, CKP, in non-volatile memory, NVM, and (3) Security - checkpoints are protected in NVM.

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.

1.3 Intermittent Computing

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. Figure 1.2 illustrates the checkpoint generation and restoration process involved when an intermittent system is powered by a solar energy harvester. 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 that focuses on efficient checkpointing techniques for intermittent devices.

Several intermittent computing techniques have been proposed, among which a majority optimize two criteria, energy efficiency and rollback minimization. The latter also ensures that the former is achieved by preventing re-execution of completed tasks. The state-of-the-art techniques use various techniques, such as architectural support [41], energy aware checkpoint calls [44], kernel-oriented design [18], task based programming and execution model [61], non-volatile processors [49], and, probabilistic algorithms [32] to obtain energy efficient checkpoints. This research focuses on security for intermittent computing.

1.4 Securing Intermittent Computing

Security for embedded devices is not a new concept. Many systems need security guarantees on user identification, network security, secure communication, secure storage, availability, etc [79]. The guarantees are usually satisfied either by cryptographic algorithms such as symmetric, asymmetric, and hash algorithm or by a combination of these algorithms in a dedicated security protocol. Also, several security architectures provide attestation and isolation in embedded environments [31, 63, 70]. These environments provide strong guarantees on the integrity of the code execution and the information leakage of data stored in memory, as long as the power is available. However, when power is removed, these environments behave as other computing systems. Hence, they may be subject to replay attacks simply by repeated execution of power-off and power-on events. Therefore, the **main research question**(main RQ) of this dissertation is formulated as:

Main RQ: How to secure embedded systems when power loss is imminent?

The main research question is divided into the following sub-questions, addressed in each chapter of this dissertation.

1.4.1 Chapter 2: Threats and Exploits

Chapter 2 aims to identify the threats introduced by intermittent computing to embedded systems. Until 2017, none of the intermittent computing proposals has considered checkpoint security [14, 17, 41, 44]. In 2017, Ghodsi et al. [32] proposed to only encrypt checkpoints which may not provide adequate protection. To date, a majority of the state-of-the-art intermittent computing techniques fail to consider the security of checkpoints. When left unprotected, the sensitive data stored in a checkpoint is available to an attacker after a power loss. This gives rise to the first research question of Chapter 2:

> RQ2.1: What are the threats introduced by intermittent computing?

We analyze the security, or lack thereof, of checkpoints in the latest intermittent computing techniques. Under the assumption that power loss is a threat vector, we identify three security vulnerabilities that are introduced by checkpoints of intermittent computing. With unsecured checkpoints, we formulate the second research question:

> RQ2.2: How to exploit the new threats in intermittent systems?

To answer RQ 2.2, we first identify sensitive data in checkpoints of a cryptographic algorithm. Then, we attack the algorithm using checkpoints to retrieve the secret key. To the best of our knowledge, Chapter 2 presents the first systematic analysis of the security threats and their exploits in the field of intermittent computing.

1.4.2 Chapter 3: Secure Design

Chapter 3 builds on the results of the previous chapter to eliminate the threats identified in Chapter 2. It is essential to secure checkpoints to ensure the security properties are maintained across power loss. With the shift from continuous execution to intermittent execution, the checkpoints become vital in maintaining overall system security. We first identify the fundamental checkpoint security objectives with the following research question.

> RQ3.1: What are the additional security requirements required in the light of emerging vulnerabilities in intermittent systems?

We performed a detailed analysis of the state-of-the-art works in checkpoint security and the lack of comprehensive security for checkpoints. We identified checkpoint security solutions using memory isolation [11, 28] and using cryptographic primitives [11, 32, 53, 100]. While isolation prevents unauthorized access of checkpoints, only cryptographic primitives encode information security properties, such as confidentiality, integrity, and/or freshness, within checkpoints. In this research, we focus on securing checkpoints using cryptographic primitives. With the security objectives in mind, we develop solutions to answer the following research question:

RQ3.2: How to secure intermittent systems using a generic framework?

We propose a secure checkpointing technique called the Secure Intermittent Computing Protocol (SICP). The proposed protocol has the following properties. First, it associates every checkpoint with a unique power-on state to prevent checkpoint replay. Second, every checkpoint is cryptographically chained to its predecessor, providing continuity, which enables the programmer to carry run-time security properties such as attested program images across power loss events. Third, SICP is atomic and resistant to power loss. Chapter 3 demonstrates a prototype implementation of SICP on an MSP430 microcontroller and performs preliminary analysis on the overhead of SICP for several cryptographic kernels.

1.4.3 Chapter 4: Configure and Optimize

The experiments in Chapter 3 revealed a considerable overhead in securing checkpoints using a cryptographic solution. We need to reduce the energy consumption of securing checkpoints and speed up the process as intermittent systems are already energy starved and need a fast wake-up time. This gave rise to the following research question on system optimization:

RQ4.1: How to improve the performance of secure intermittent systems?

In Chapter 4, we propose to configure security properties for different sections of a checkpoint. The main idea is to further whittle the security objectives for different sections of checkpoints to minimize the checkpoint security requirements and consider other security requirements as add-on features. We propose to achieve this using configurable security levels for intermittent systems that start with minimal security requirements and adds other security features in an ad-hoc manner.

1.4.4 Chapter 5: Evaluate and Benchmark

Finally, Chapter 5 will study the two-way dependency between the application and checkpoint security. On the one side, we believe the overhead of securing checkpoints, including time and energy, affects the actual forward progress of the intermittent application. And on the other side, the application also plays a major role in deciding the security requirements of checkpoints which affect the overall overhead of securing checkpoints. The experiments in Chapter 5 will address the following research question:

RQ5.1: What are the factors that affect the performance of secure intermittent systems?

In the first experiment, we demonstrate the effect of securing checkpoints on the duty cycle of a secure key exchange algorithm. In the second experiment, we curate a list of benchmarks to quantitatively evaluate the overhead of application, intermittent computing, and configurable secure intermittent computing. This chapter systematically evaluates the impact of application and other factors on the performance of secure intermittent systems.

1.5 Reading Guide

The main chapters of this dissertation (Chapters 2 - 5) have been written as individual papers, each chapter can be read as its own. As a result, there may be some overlap in related work sections of the main chapters. An adapted version of Chapter 2 and Chapter 3 has been published in the *Journal of Hardware Architecture and System Security*. An adapted version of Chapter 4 and Chapter 5 is under review for publication in another journal. The results in this dissertation have been presented at various conferences and workshops including *International Conference on Security, Privacy, and Applied Cryptography Engineering* (2018), *International Symposium on Hardware Oriented Security and Trust* (2019), *Design, Automation & Test in Europe* (2019), and *(*2019).

Chapter 2

Exploiting Vulnerabilities in Intermittent Computing

2.1 Introduction

Traditional embedded systems are conceived from the viewpoint that power is plentiful and that power will only be fully removed when all tasks are completed. The power management is adjusted to the needs of the application or to the computing load. 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 availability of energy, the device is powered on/off, leading to an intermittent operation. In intermittent systems where the power source is unreliable and limited, the application adapts to the available power, including seamless and transparent turn-off and turn-on.

Intermittent systems use scavenged or harvested energy sources, which provide a nearly inexhaustible energy supply with limited and unreliable power delivery (think of a solar cell). Depending on the energy harvesting source, the power level can be as low as a few microwatt. Through power conditioning, harvested energy is stored in an energy buffer, such as a supercapacitor, which in turn has limited capacity and which may overflow. This makes continuous operation of intermittent systems- virtually impossible; at some point, the energy buffer runs out. However, by saving critical system state as a *checkpoint*, system operation can continue across power loss. The checkpoint generation is either triggered by a system call in volatile processors or automatically triggered by a power interrupt in non-volatile processors [49]. Non-volatile processors store a majority of their data in non-volatile memory and place their system data, such as registers, in volatile memory. They only have to back up the volatile state in non-volatile logic, thus have the advantage of instant state restore on power-up with negligible energy and time overhead.

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 [40, 84]. 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 [66]. Sneak -path encryption (SPE) was proposed to secure non-volatile memory using a hardware intrinsic encryption algorithm [47]. 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 [24]. 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 [41, 44, 78]. 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 chapter, 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 [96]. 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 chapter:

- 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 Section 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 Section 8.

2.2 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 possesses the objective to attack the target algorithm. In both the cases, the attacker has the following capabilities.

- The attacker can access the memory via traditional memory readout ports or employ sophisticated on-chip probing techniques [40, 84], 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 Section 2.5 using Figure 2.1

- 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.

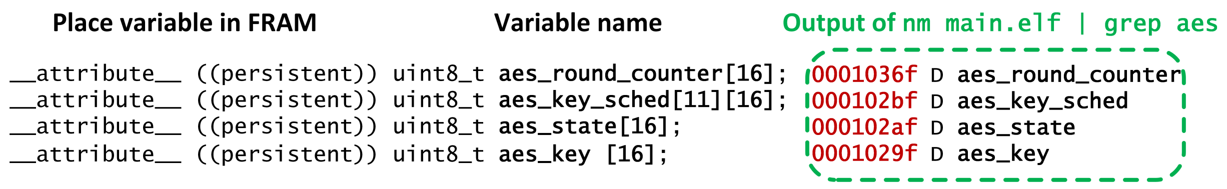
2.3 Security Vulnerabilities of Unsecured Checkpoints

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 chapter 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 [32], 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.

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.



```
__attribute__ ((persistent)) uint8_t aes_round_counter[16];
__attribute__ ((persistent)) uint8_t aes_key_sched[11][16];
__attribute__ ((persistent)) uint8_t aes_state[16];
__attribute__ ((persistent)) uint8_t aes_key [16];
```

```
0001036f D aes_round_counter
000102bf D aes_key_sched
000102af D aes_state
0001029f D aes_key
```

Figure 2.1: 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`.

2.4 Experimental Setup

We used TI's MSP430FR5994 LaunchPad development board. The target device is equipped with 256kB of FRAM which is used to store the checkpoints. We implement TI's software AES128 library on MSP430FR5994 as the target application running on the intermittent device. Figure 2.1 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 2.1 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 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 chapter.

2.4.1 CTPL

TI introduced CTPL [96] which provides a checkpoint on-demand solution for intermittent systems, similar to QuickRecall [44]. It defines dedicated linker description files for all its MSP430FRxxxx devices that allocates all the application data sections in FRAM

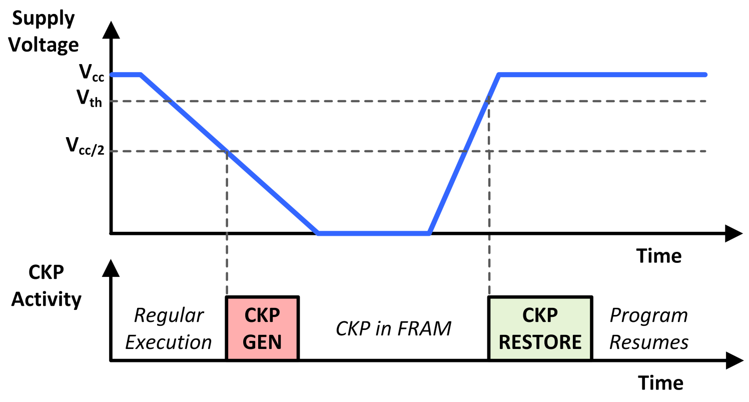


Figure 2.2: Principle of operation of CTPL, checkpoint(CKP) generation and restoration based on the supply voltage, V_{cc}, and its set threshold voltage, V_{th}

and allocates a storage location to save volatile state information. Figure 2.2 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 Figure 2.2. 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.

Checkpoint generation: Call to `ctpl_enterShutdown()` function saves the volatile state in three steps, as shown in the bottom of Figure 2.3. In the first step, the volatile periph-

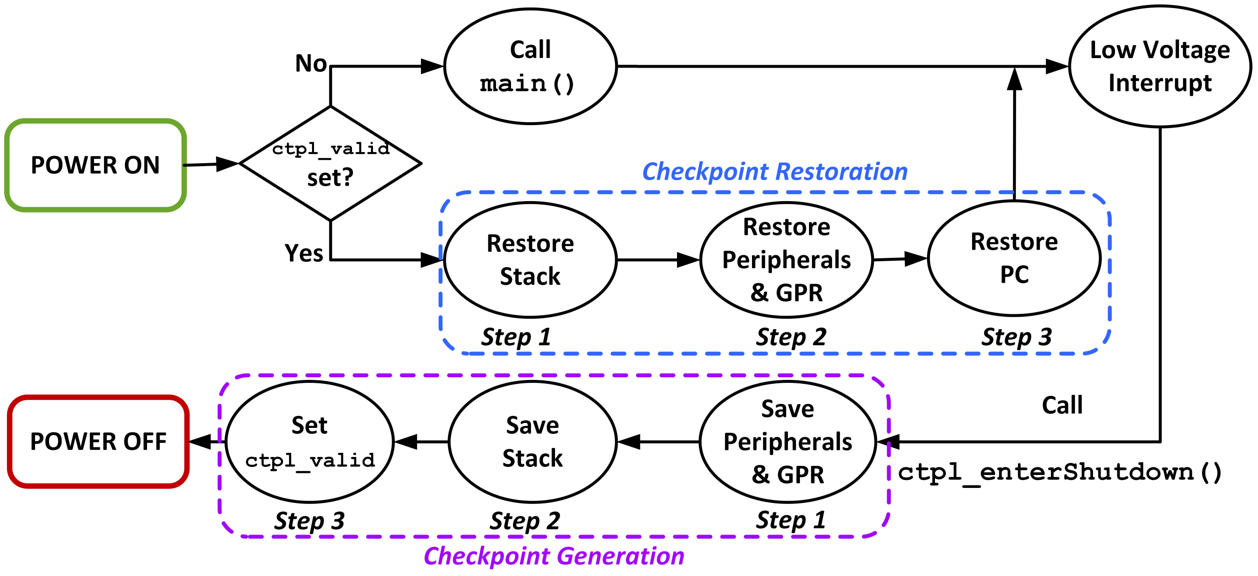


Figure 2.3: CTPL checkpoint generation and restoration flowchart

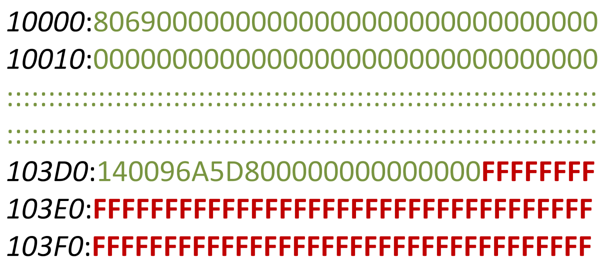
eral state, such as a timer, comparator, ADC, UART, etc., and general purpose registers (GPRs) are stored in the non-volatile memory. The second and third step are programmed in assembly instructions to prevent mangling the stack when it is copied to the non-volatile memory. In the second step, the watchdog 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 Figure 2.3. 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.

2.5 Exploiting CTPL's Checkpoints

In this section, we 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. 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, [40, 84], can also be utilized to deploy our attack on AES when JTAG interface is disabled or unavailable.



```
10000:80690000000000000000000000000000
10010:00000000000000000000000000000000
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
103D0:140096A5D80000000000000000FFFFFFFF
103E0:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
103F0:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

Figure 2.4: Memory dump of FRAM, where the checkpoint begins from 0x10000 and ends at 0x103DB

2.5.1 Capabilities of a knowledgeable attacker

Armed with the information about CTPL and the target device, a knowledgeable attacker analyzes the 256kB 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 betweem 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 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 256kB of FRAM was initialized, and the majority of the FRAM was filled with 0xFFFF, as shown in Figure 2.4. 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.

2.5.2 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 of 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,

```
diff md_001.txt md_010.txt
43,44c43,44
< :102A2:FF709AE3CC8FDB755E4FA44F2471AA09
< :102B2:3A0A68F305143A9F01A4CECE2084360O
---
> :102A2:FF709AE3CC8FDB755E4FA44F2471AA1B
> :102B2:2A1B515A23F3B8C2995274B82AE25F00

diff md_003.txt md_006.txt
43,44c43,44
< :102A2:FF709AE3CC8FDB755E4FA44F2471AA7C
< :102B2:4F95825EE955979F01A4CECE2084360O
---
> :102A2:FF709AE3CC8FDB755E4FA44F2471AA63
> :102B2:50B4BC5EE955979F01A4CECE2084360O
```

```
diff md_001.txt md_008.txt
43,44c43,44
< :102A2:FF709AE3CC8FDB755E4FA44F2471AA09
< :102B2:3A0A68F305143A9F01A4CECE2084360O
---
> :102A2:FF709AE3CC8FDB755E4FA44F2471AA64
> :102B2:57BDB25EE955979F01A4CECE2084360O

diff md_001.txt md_007.txt
43,44c43,44
< :102A2:FF709AE3CC8FDB755E4FA44F2471AA09
< :102B2:3A0A68F305143A9F01A4CECE2084360O
---
> :102A2:FF709AE3CC8FDB755E4FA44F2471AA68
> :102B2:5BA9AA5EE955979F01A4CECE2084360O
```

Figure 2.5: 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

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 [26]. 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 Figure 2.5. 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 Figure 2.5, 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.

2.6 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.

2.6.1 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:

$$N_{KeyGuess} = L_{memory} - L_{key} + 1 \tag{2.1}$$

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 equation 2.1, 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 256kB of FRAM, which is 256,000 bytes long. Using equation 2.1, 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.

2.6.2 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 [13] and single byte faults [7]. A majority of these methods require dedicated hardware support in the form of laser [7] or voltage glitcher [16] 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 chapter, 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 [34] and [30] respectively.

Inducing single bit faults: To implement the single-bit DFA described in [34], 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 [34]. 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 [30], 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 [30] 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.

2.6.3 Replaying checkpoints to enable side channel analysis

The secret key of AES can also be extracted by using differential power analysis (DPA) [50]. 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.

2.6.4 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 [30, 34], 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 chapter 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 chapter are also effective against other cryptographic algorithms and security features, such as control flow integrity protection [27] and attestation solutions [31], that maybe implemented on an intermittent device. Thus, unprotected checkpoints undermine the security of the online protection schemes incorporated in intermittent devices.

2.7 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 chapter, 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 to retrieve the secret key. This calls for intermittent computing designs that address the security pitfalls introduced in this chapter. 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.

Chapter 3

Securing Intermittent Computing

3.1 Introduction

Computers including servers, personal computers (PCs), laptops, and embedded devices, run on electric power, which is typically supplied by the grid. Power loss, a fact of life, is a short-term or long-term shortage of power which causes computer shut downs. Upon power loss, the device transitions from ON-state to OFF-sate, losing its volatile computer state. Upon the next power-up, it transitions to ON-state and re-initializes the volatile state, thus power loss re-initializes the system on every power-up. The transition between ON, OFF, and ON-state is called power transition. The computer copes with power loss by storing checkpoints of the intermediate volatile state in non-volatile memory, illustrated in Figure 1.2. Non-volatile memory ensures that checkpoints remain persistent across power transitions. Upon power up, the computer is restored to the most recent checkpointed state and resumes its tasks.

In this chapter, we focus on the power transitions of a secure embedded system. Energy harvesting technology converts ambient energy to electrical energy, which is sufficient to power resource constrained embedded devices. Figure 1.2 illustrates a device powered by a solar energy harvester. Since the availability of solar energy depends on the weather and time of the day, a solar energy harvester is a transient power source. Transient power supplies do not provide continuous power which causes power loss in embedded systems. To cope

Table 3.1: Comparison of the essential checkpoint security properties among various state-of-the-art related work in the field of both emebdded devices and conventional computer

Type of solution	**Related Work**	**Essential properties**						**Target Platform**
		C	**I**	**Auth**	**F**	**Cont**	**Atom**	
Intermittent computing	[14, 17, 41, 44]	-	-	-	-	-	-	Embedded device
	Ghodsi [32]	✓	-	-	-	-	-	
NVM memory encryption	iNVM [24], SPE [47]	✓	-	-	-	-	-	Conventional computer
State continuity	ICE [91]	✓	✓	-	✓	✓	-	Conventional computer with protected module
	Ariadne [90]	✓	✓	-	✓	✓	✓	
	Memoir [73]	✓	✓	-	✓	✓	✓	
Secure checkpoints	SECCS [100]	✓	✓	✓	-	-	-	Embedded devices
	Asad [11]	✓	✓	-	-	-	-	
	SICP(this work)	✓	✓	✓	✓	✓	✓	

C: Confidentiality, I: Integrity, Auth: Authenticity, F: Freshness, Cont: Continuity, Atom: Atomicity.

with power loss, the device is equipped with non-volatile memory. Although the device's non-volatile memory retains its data during power-off, the volatile state information is lost. Non-volatile memory by itself is insufficient to ensure forward progress of the application [61]. Intermittent computing is a stateful power transition technology, where the device stores a snapshot of the volatile state information in non-volatile memory, as a checkpoint (CKP). The checkpoint is used to restore the device to the last known state to ensure forward progress of the application. The state-of-the-art intermittent computing techniques provide efficient checkpoint generation and restoration solutions to ensure forward progress with minimum overhead [41, 61, 65, 101]. As the checkpoints contain intermediate state of the device, they must be secured to protect power transitions.

Related Work Table 3.1 compares the essential properties of some of the latest work related to checkpoints. So far, none of the intermittent computing proposals has considered checkpoint security [14, 17, 41, 44], except one [32], which only considers confidentiality and does not detect checkpoint replay. In-band memory encryption techniques have been proposed for conventional computers [24, 47], which introduce a constant encryption overhead that is less suited for embedded devices.

Current conventional computers are equipped with module isolation mechanisms, such as Intel SGX and ARM TrustZone, that also require state continuity guarantees in case of system crashes and power loses. ICE [91], Ariadne [90] and Memoir [73] were designed to provide state continuity to these computers. Although these solutions guarantee most of the essential properties, they are not designed for resource constrained embedded devices.

SECCS, a secure context saving solution, only provides confidentiality and integrity of checkpoints in non-volatile memory using a hardware module [100]. It does not consider replay of checkpoints or availability of the intermittent device, similar to Asad et al. [11]. As a result, both SECCS and Asad et al. do not ensure freshness and authenticity of checkpoints and is not an atomic solution.

Figure 1.2 illustrates the three facets of power transitions - cause, statefuleness, and security. In general purpose computers, such as servers, PCs and laptops [35, 67], all three facets of power transitions are on-going research problems. Whereas in embedded systems, only the cause [58, 89] and statefulness [41] are commonly explored. Although security for embedded systems is an on-going research problem [70], the security of power transitions is widely ignored. In this chapter, we highlight the need for secure power transitions in embedded systems through the following contributions:

- We study the security vulnerabilities introduced by stateful power transitions and analyze the need for secure power transitions.
- We propose the Secure Intermittent Computing Protocol (SICP) to overcome these vulnerabilities. We describe a real-life application that requires checkpoint security, which can benefit from SICP.

- We quantify power transitions in embedded systems by computing the overhead of checkpoint generation and restoration process, and the overhead incurred to secure stateful power transitions. We demonstrate that secure and stateful power transitions are expensive but achievable in embedded systems.

Organization The rest of the chapter is organized as follows. Section 3.2 provides a brief background on the different facets of power transition and their effects on embedded systems. Section 3.3 discusses our attacker model, locates checkpoint vulnerabilities, and provides a set of security requirements for checkpoints. Section 3.4 proposes SICP to satisfy these security requirements followed by our implementation of SICP in Section 4.4. Section 3.6 evaluates SICP by introducing the need for checkpoint security to a real-life application and by studying the overhead of statefulness and security in power transition and followed by our conclusions in Section 3.7.

3.2 Background in Power Transitions

In this section, we define the different facets of power transitions and analyze their effects on embedded systems. The three facets of power transition are defined as follows:

1. Cause: The root cause of power loss helps identify the frequency, period of power loss, and other characteristics which help design coping mechanisms for the computer system.

2. Statefulness: A stateful power transition is aware of the intermediate state of the computer system. Through statefulness, the computer maintains its state during power loss which is used in future computations. It ensures the forward progress of the application.

3. Security: The security of a power transition is the guarantee that the state of the computer system is protected from data corruption and unauthorized access even during power loss. It preserves the security features of both the device and application across power loss.

We analyze the problems introduced by transient power supplies, describe the use of statefulness to cope with these problems and demonstrate the need for security in stateful power transitions.

3.2.1 Cause

Energy harvesters extract energy from ambient energy sources, such as heat [89], vibration [104], and radiation [60], and convert it to electrical energy to power embedded devices. Since the ambient energy is not always available, energy harvester provide intermittent power supply to embedded devices. The embedded device turns-on and turns-off based on the availability of input energy. Conventionally, after each power cycle the device is reinitialized and loses the progress made during the previous power on state, restarting the application every time. Stateful power transitions are needed to avoid re-initialization after every power loss.

The intermittent computing model, a stateful power transition technique, was introduced to guarantee forward progress of long-running applications when powered by an intermittent power supply. All the state information necessary to restore the device is stored as a checkpoint in non-volatile memory. A checkpoint consists of the system state, such as processor registers, peripheral registers, and application state, such as stack, heap, and developer defined variables that are required to resume program execution. After a power cycle, the device is restored to the last known checkpointed state. Several intermittent computing techniques have been proposed, among which a majority optimize two criteria, energy efficiency and rollback minimization.

Irrespective of the checkpointing technique in use, the device transitions through two states, ON-state and OFF-state. During the ON-state, the device performs its regular tasks. It may employ protection features such as control flow integrity [27], attestation and isolation [70], and protection against cold-boot attacks [36]. The variables required to implement these security features must also be checkpointed to ensure the continuation of these security properties in future ON-states. During the OFF-state, the checkpoint remains in non-volatile memory. The checkpoint contains the intermediate state of the application, which may be a cryptographic algorithm, and the critical settings of the security features employed during ON-state, such as kernel privileges and memory access rights. A majority of the intermittent computing techniques store their checkpoints as plaintext in non-volatile memory. A few techniques explore security in power transitions [32, 100] but they do not provide a comprehensive security solution. The existing secure power transition solutions from general purpose computers [21, 22, 35] cannot be used in embedded systems because they were not designed for resource constrained devices.

3.3 Problem Description

Checkpoints, which are generated to provide stateful power transitions, introduce vulnerabilities to an embedded device which may otherwise be secure when it is powered on. In this section, we define our attacker model, describe the risks introduced by unprotected checkpoints, and list a set of minimum security requirements to protect power transitions against the assumed attacker model.

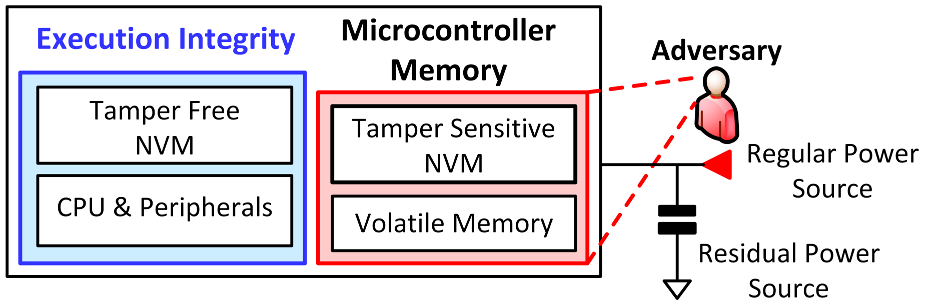


Figure 3.1: The architectural assumptions and memory model for SICP illustrating the assumed attacker model with two capabilities - (1) control power supply to the device and (2) view and modify tamper sensitive non-volatile memory during power-off periods.

3.3.1 Attacker model

The attackers aims to gain useful information from the intermittent execution model. We define an attacker model with the following capabilities, illustrated in Figure 3.1, to study the security vulnerabilities introduced by checkpoints.

1. The attacker has complete control over the power supplied to the device. The attacker can arbitrarily stop the application on the target device, for example, the attacker can tamper with the energy harvester input to control the input to the target device. The aim of the attacker is not to completely stop the application on the target device, but to stop the target device at strategic points in the application to gain information from the checkpoints. Thus, denial of service by cutting off power supply is out of scope of this attacker model.

2. The attacker has access to the majority of the device memory when it is powered off. The attacker can read from and write to the unprotected non-volatile memory, which we call tamper-sensitive non-volatile memory. In this scenario, even though the device must be powered on to access the contents of memory, the CPU is still not powered-on, i.e, the processor is in idle state. For example, the attacker can access

the memory by providing read/write commands to Direct Memory Accessc(DMA) via debug probes. Since DMA is independent of the processor, the attacker need not power-on the processor to access memory [75].

3. The attacker cannot tamper with secure non-volatile memory and residual capacitors on board. The secure non-volatile memory is used to store the secret key and parts of secure checkpoint. A few of the on-board capacitors are used as residual energy source for certain protocol operations such as atomic write and erase operations.

We assume that the device is equipped with a tamper-free non-volatile memory, which is secure from the assumed attacker model. This requirement can be satisfied by using an off-the-shelf microcontroller with secure non-volatile memory, such as Maxim's ZA9L1 [3]. For example, the secure memory may only be accessible from authorized code and unauthorized access may lead to zeroization of secure memory. In the event of power loss, we assume that the device is equipped with residual capacitance, which provides sufficient power to the device to complete a 128-bit write without interruptions. We assume that the device is physically protected from the attacker, including the CPU and other components on-board. The attacker cannot access the device memory during ON-state, the volatile and non-volatile system states are inaccessible to the attacker when the device is powered on. We assume that the device's execution integrity and memory protection during power-on states are guaranteed by a protected embedded software execution environment [70]. The mitigation of side channel and fault injection attacks on the checkpointing system are beyond the scope of this work.

3.3.2 Checkpoint vulnerabilities

Non-volatility of persistent memory compromises the privacy of unsecured persistent data. The state-of-the-art non-volatile memory protections are not designed for resource constrained devices [47, 94]. The state-of-the-art intermittent computing techniques also fail to secure their checkpoints. Checkpoints consist of the volatile and non-volatile state of a device, which may contain sensitive data. When left unsecured, they introduce the following vulnerabilities to an intermittent system.

- *Checkpoint snooping*: The attacker can read the non-volatile memory, and in turn read the checkpoints to extract sensitive information stored in them as the checkpoints are stored as plaintexts. Non-volatile data, which is otherwise private during power-on, is now open to attackers in checkpoints. The attacker can study the checkpoints to identify the location of sensitive information [83]. While checkpoint encryption may provide protection against snooping [32], it does not protect against the other vulnerabilities.

- *Checkpoint spoofing*: The state-of-the-art intermittent computing techniques simply restore a checkpoint, if one exists, without checking its integrity. With the knowledge of the location of sensitive variables, the attacker can spoof checkpoints by modifying them in non-volatile memory. Unknowingly, the device restores itself with a modified checkpoint from where it resumes execution in an attacker controlled sequence. Encrypted checkpoints [32] are also vulnerable to spoofing as they do not guarantee integrity. The attacker can modify an encrypted checkpoint, which may not correspond to a valid checkpointed state upon decryption. When the device is powered up, it is restored with the decrypted modified encrypted checkpoint, which may lead to a system crash.

- *Checkpoint replay*: The attacker can combine snooping and spoofing to replay checkpoints. The attacker can store a copy of all the checkpoints of an intermittent system, where each checkpoint corresponds to a state of the application, to create a pool of checkpoints. The state-of-the-art intermittent computing techniques do not check if the checkpoint to be restored is indeed the latest checkpoint, which enables checkpoint replay. The attacker can overwrite the current checkpoint with any checkpoint from their pool; upon power-up, the device is restored to a stale state. A checkpoint security solution which only protects checkpoint confidentiality and integrity, such as SECCS [100], will not detect checkpoint replay.

3.3.3 Exploiting unsecured checkpoints

The attacker can exploit these vulnerabilities to gain access to sensitive information about the application on the device. If a device is programmed with a cryptographic algorithm, such as Advanced Encryption Standard(AES) [26], the application variables must be included in its checkpoint to ensure forward progress of the algorithm in the event of power loss. The attacker can identify the sensitive variables in a checkpoint [83], such as the intermediate state and round counter of AES. The ability to spoof checkpoints enables the attacker to replace sensitive variables of AES with attacker controlled variables and extract the secret key using cryptanalysis.

Checkpoint security is essential to ensure that the security properties of ON-states are maintained across power transitions, without any compromise. The continuous execution paradigm is shifting to an intermittent execution paradigm, which makes checkpoints an integral part of the execution environment. The existing secure software execution environments are only designed for ON-states based on the assumption that the power supply is continuous [27, 31, 70]. They propose to restart their system, including the security modules and features, when they encounter a power failure. These assumptions do not apply to a system powered by a transient power supply. Secure software execution must consider the security of both its ON-state and OFF-state, which includes checkpoint security.

3.3.4 Checkpoint Security Requirements

Although the security requirements may vary depending on the application and device, we must consider the following as a set of minimum requirements to overcome the vulnerabilities discussed above.

- *Information security*: The checkpoint's confidentiality, integrity, authenticity, and freshness must be ensured to protect against checkpoint snooping, spoofing, and replay.
- *Availability*: The checkpoint generation and restoration process must be atomic to ensure a valid checkpoint is always available. This guarantees that the checkpoints will not be corrupted even if a power loss occurs during the checkpointing process.
- *Continuity*: Secure application continuity maintains the order of checkpoints, to provide assurance that the device is at the current state because it executed the previous states without any attacker intervention.

3.3.5 Architectural Assumptions

Secure and stateful power transitions require certain architectural features and protection guarantees, illustrated in Figure 3.1. The device must have three types of memory. First, volatile memory to store the runtime program state, which is erased upon power loss. Second, tamper sensitive non-volatile memory, which does not possess any tamper resistance. Third, tamper-free non-volatile memory, which is secure against the assumed attacker model. The size of tamper free memory must be minimized to reduce hardware cost and complexity. We only place necessary variables in tamper-free memory, including the secret key and nonce, instead of placing the entire secure checkpoint in it. The rest of the secure checkpoint is placed in the tamper sensitive non-volatile memory, which is unprotected.

Apart from the different types of memory, the device must have a residual power source to provide a small, finite source of energy. For example, an on-chip or on-board capacitor may act as a residual source to power the device for a small period even after the main power supply is powered-off. Since power loss is considered a threat, sensitive variables must be wiped as soon as the device encounters a power loss. We assume that the residual power source is sufficient to wipe sensitive variables and to finish writing a 128-bit value in non-volatile memory. Since the device is physically protected from the attacker, the assumed physical protection also extends to the residual source.

3.4 Secure Intermittent Computing Protocol

Checkpoint security is essential, without which the security features from ON-state are lost during OFF-state. Intermittent computing techniques only ensure the forward progress of the application, the continuity of the security properties require a set of rules to detect and prevent tampering. This introduces a need for a protocol or a frame of reference to describe and achieve the security requirements discussed in Section 3.3.4.

We define the Secure Intermittent Computing Protocol (SICP) to protect the checkpoint vulnerabilities introduced in Section 3.3.2 and to ensure forward progress of the application and continuity of security properties. SICP defines a set of rules among the different states of

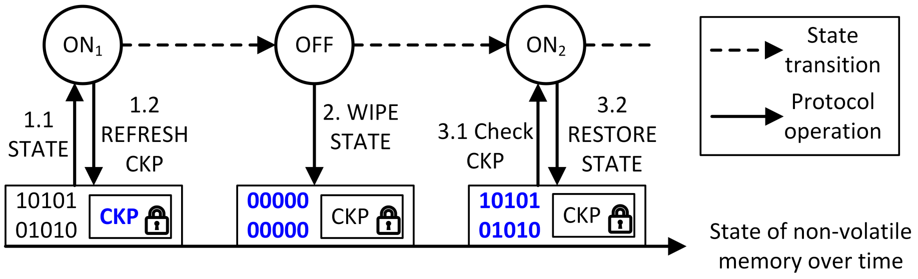


Figure 3.2: A protocol scenario for secure power transitions, depicting a sequence of ON-states, OFF-states, and the corresponding state of non-volatile memory of the device. The protocol provides rules for (1) creating secure checkpoints, *CKP*, during *ON₁* and for (3) restoring an unmodified *CKP*. It also ensures the protection of plaintext state by (2) overwriting it with zeros upon power loss.

the device, illustrated in Figure 3.2. The non-volatile memory, which holds the checkpoints, is the prover and the device verifies the validity of these checkpoints. During power-on, the device creates a secure checkpoint and stores it in non-volatile memory (Step 1). After a power cycle, the device verifies if the checkpoint to be restored is indeed the latest and unmodified checkpoint (Step 3). With SICP, the device can differentiate between a malicious and valid checkpoint in memory. It detects malicious checkpoints and prevents restoring the device to an attacker controlled state.

3.4.1 Satisfying the security requirements

We start with a device that has gone through *factory_reset()* which restores the device to manufacturer settings and programs the tamper-free non-volatile memory with a secure key, K. With the unsecured checkpoint, $STATE$, which contains the application and microcontroller data, we create a secure checkpoint in several steps, illustrated in Figure 3.3.

First, the freshness requirement is satisfied by associating each $STATE$ with a nonce, R_i, which is stored in tamper-free non-volatile memory. *nonce()* generates a unique and fresh R_i. Second, the confidentiality, integrity and authenticity requirements are satisfied by encrypt-

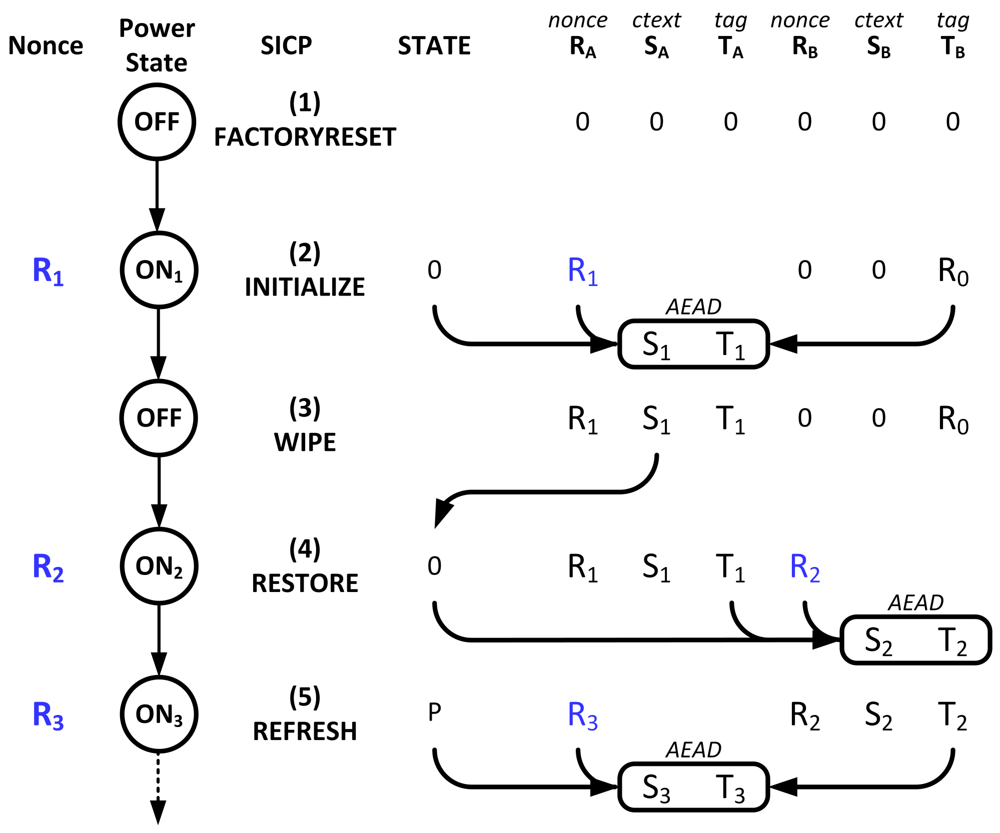


Figure 3.3: An example SICP scenario. (1) The system is cleared by the *factory_reset()* operation. (2) A fresh nonce, R_i is associated with each power-on state. The first valid state save packet, SS_1, is created by `INITIALIZE`. On power loss, (3) `WIPE` clears the volatile $STATE$ and upon subsequent power up, (4) `RESTORE` validates the latest state save packet, SS_1, restores the program state, and generates a new state save packet SS_2. (5) During program execution, `REFRESH` is called to create a new checkpoint SS_3, overwriting the oldest state save packet, SS_1.

ing $STATE$ and R_i using Authenticated Encryption with Associated Data (AEAD) [81]. $AEAD_{encr}()$ takes the plaintest $STATE$, R_i, and the non-confidential associated data as input to generate the encrypted checkpoint, S_i. $AEAD_{auth}()$ generates an authentication tag, T_i, over the newly encrypted checkpoint along with the nonce and associated data[1]. After a power cycle, if a valid authentication tag exists, it decrypts S_i using $AEAD_{decr}()$. If the authentication tag check fails, *abort()* is called to raise a violation of the protocol. At a

[1]The encryption and tag calculation in AEAD operations are separated here to provide clarity in protocol operations

Algorithm 1 `INITIALIZE`

```
Require: $K$
1: $Q \leftarrow nonce()$
2: $T_B \leftarrow nonce()$
3: $STATE \leftarrow 0$
4: $R_A \leftarrow Q$
5: $S_A \leftarrow AEAD_{encr}(STATE, T_B, R_A, K)$
6: $T_A \leftarrow AEAD_{auth}(S_A, T_B, R_A, K)$
```

minimum, *abort()* must either halt the device or clear the device memory and restart it. A secure checkpoint is a tuple of S_i, R_i, and T_i, which is called a state save packet, SS_i.

Third, the atomicity requirement is satisfied by storing the state save packets in a two-state buffer, SS_A and SS_B. They are updated in an alternating manner to ensure one packet is kept valid at all times. At a given point of time, the non-volatile memory will contain the latest packet, SS_i, and the previous packet, SS_{i-1}, illustrated in Figure 3.3. Fourth, the continuity requirement is satisfied by tag-chaining, which is the process of cryptographically chaining the authentication tags of the checkpoints in chronological order. It is achieved by using the authentication tag from the previous packet, T_{i-1} as associated data to generate the latest packet, SS_i. For example, in Figure 3.3, T_1 is used to compute SS_2, from which T_2 is used to compute SS_3. The authentication tags protect the integrity and authenticity of checkpoints as well as its chronological order.

3.4.2 Protocol

We define SICP as a collection of four algorithms described below.

INITIALIZE : The device is initialized with the first packet, SS_1, with Algorithm 1. Upon power-up, INITIALIZE is called if the device has gone through a *factory_reset()*, which is identified by a unique reset memory pattern. INITIALIZE is called only once to create

Algorithm 2 Original `REFRESH` and `RESTORE`

```
Require: $K, STATE, S_i, R_i, T_i$, where $i \in \{A, B\}$
         $operation \in \{\texttt{REFRESH}, \texttt{RESTORE}\}$
 1: $Q \leftarrow nonce()$
 2: if $T_A = AEAD_{auth}(S_A, T_A, R_A, K)$ then
 3:     if $operation$ = RESTORE then
 4:         $STATE \leftarrow AEAD_{decr}(S_A, T_A, T_B, R_A, K)$
 5:     end if
 6:     $R_B \leftarrow Q$
 7:     $S_B \leftarrow AEAD_{encr}(STATE, T_A, R_B, K)$
 8:     $T_B \leftarrow AEAD_{auth}(S_B, T_A, R_B, K)$
 9: else
10:     if $T_B = AEAD_{auth}(S_B, T_A, R_B, K)$ then
11:         if $operation$ = RESTORE then
12:             $STATE \leftarrow AEAD_{decr}(S_B, T_B, T_A, R_B, K)$
13:         end if
14:         $R_A \leftarrow Q$
15:         $S_A \leftarrow AEAD_{encr}(STATE, T_B, R_A, K)$
16:         $T_A \leftarrow AEAD_{auth}(S_A, T_B, R_A, K)$
17:     end if
18: else
19:     $abort()$
20: end if
```

SS_1, which is stored in buffer SS_A. Since the first packet has no previous authentication tag to be used as associated data, a nonce is used as associated data, T_B. This ensures a unique chain of tags are generated after every *factory_reset()*. Next, *STATE*, where the plaintext checkpoint is collected, is zeroized to overwrite the reset memory pattern to prevent future calls to INITIALIZE. A valid state save packet, SS_A, is created by encrypting and authenticating *STATE*, R_A, and T_B using AEAD to generate S_A and T_A. A state save packet, SS_i, is valid if it satisfies two conditions. First, its nonce, R_i, must match the nonce used in AEAD operations. Second, its associated data in the AEAD operations must match the authentication tag of the previous state save packet. It ensures only one packet is valid between the two buffers, SS_A and SS_B.

REFRESH : Algorithm 3 defines both the secure checkpoint generation and restoration process, as they involve similar cryptographic operations with the difference listed on line

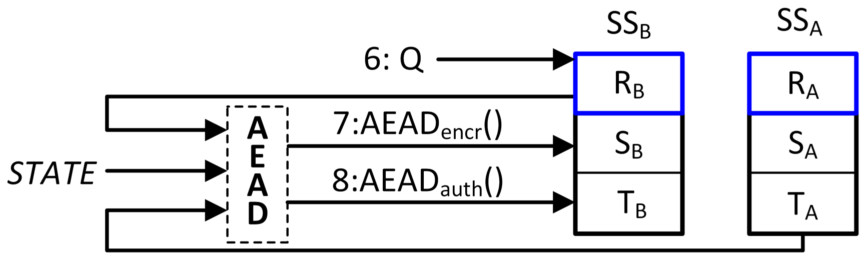


Figure 3.4: An example REFRESH of state save packet, SS_B, based on Algorithm 3. (1) Update R_B with the latest nonce, Q (line 6), (2) encrypt the checkpoint with the nonce and authentication tag from previous packet, T_A, and update S_B (line 7), and (3) last, update the authentication tag, T_B, which invalidates SS_A and validates SS_B as the most recent valid packet (line 8).

4 and 12. During power-on, REFRESH is called to generate the latest state save packet. It determines which is the valid buffer, between SS_A and SS_B to update the alternate buffer. For example, when REFRESH is called, if SS_A is valid, line 2 in Algorithm 3 is true. Correspondingly, SS_B is updated with the latest checkpointed state by first updating R_B and then S_B, as illustrated in Figure 3.4. SS_A remains valid until T_B is updated. As soon as T_B is updated with the latest authentication tag, in line 8, SS_A is invalidated and SS_B is the latest valid packet. This update to the authentication tag, T_B in line 8 and T_A in line 16, makes REFRESH atomic. SICP makes an explicit assumption that this tag update is an atomic operation. This assumption is satisfied using the residual power source, explained further in Section 3.5.4.

RESTORE : RESTORE is called upon every power-up, except immediately after a *factory_reset()*, to decrypt and restore the most recent valid *STATE* of the device. The authentication tags of both the buffers are checked to identify the valid packet. If both authentication tag checks fail, *abort()* is called to indicate checkpoint tampering, which prevents restoring the device with a malicious state.

If the authentication tag check is passed on either line 2 or 10 in Algorithm 3, a valid state save packet exists which is decrypted and used to restore the device *STATE*. RESTORE documents each power-on event in the sequence of checkpoints by generating a new state save packet upon every power-up. For example, if SS_A is valid, S_A is decrypted and restored in *STATE*. SS_B is updated with this *STATE*, new nonce, Q and T_A. Now, SS_B is made valid, invalidating SS_A. SICP ensures that every power cycle is documented in the series of checkpoints.

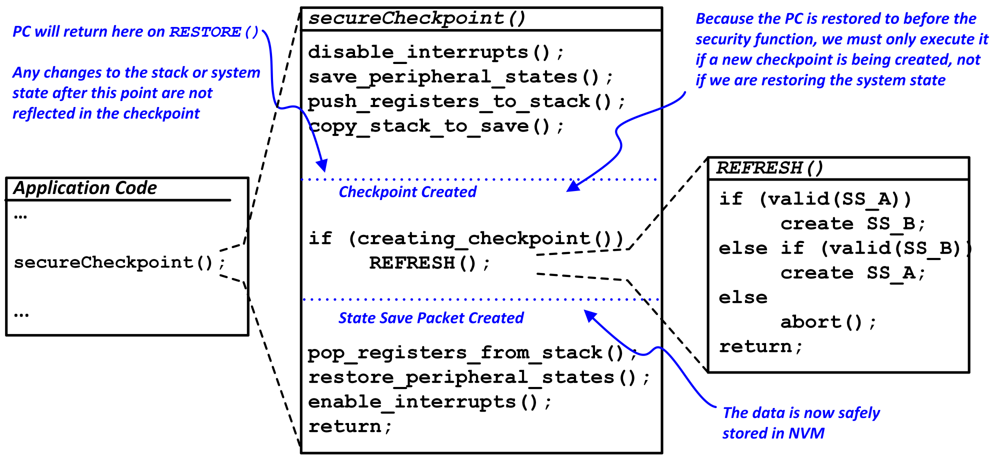

PC will return here on RESTORE()
Any changes to the stack or system state after this point are not reflected in the checkpoint
Application Code
...
secureCheckpoint();
...
secureCheckpoint()
disable_interrupts();
save_peripheral_states();
push_registers_to_stack();
copy_stack_to_save();
Checkpoint Created
if (creating_checkpoint())
REFRESH();
State Save Packet Created
pop_registers_from_stack();
restore_peripheral_states();
enable_interrupts();
return;
Because the PC is restored to before the security function, we must only execute it if a new checkpoint is being created, not if we are restoring the system state
REFRESH()
if (valid(SS_A))
create SS_B;
else if (valid(SS_B))
create SS_A;
else
abort();
return;
The data is now safely stored in NVM

Figure 3.5: The control and data flow for the creation of a checkpoint and subsequent state save packet. `REFRESH()` is only called when a checkpoint is created.

WIPE Power loss is an adversarial event, based on our attacker model. WIPE must be called as soon as the device detects a power loss to clear sensitive information. It wipes all transient information, such as program variables, stored as plaintext using the residual power source in two steps. First, *STATE* is overwritten with zeros to clear persistent plaintext information. Second, volatile memory is also wiped to prevent cold boot style attacks []. The residual power source must have sufficient power to completely wipe transient information and maintain the confidentiality of checkpointed data.

3.5 Implementation

In this section, we describe our choice of target device, stateful power transition technique, and several design choices and device specific features used in implementing SICP.

3.5.1 Target device

The embedded device used with energy harvesters plays an important role in utilizing the harvested energy and is selected based on several criteria. First, on-chip non-volatile memory is required to store checkpoints. The use of off-chip non-volatile storage in the absence of on-chip non-volatile storage is not a secure solution, as the communication to off-chip memory and the memory itself is vulnerable to attackers as it can be easily monitored/removed. Second, the device must consume low power to judiciously use the available resources. The choice of device determines the overhead incurred by secure and stateful power transition.

We implement SICP on Texas Instruments'(TI) MSP430FR5994 Launch Pad Development Kit to demonstrate the feasibility of and to evaluate secure and stateful power transitions. We chose TI's MSP430FR5994 for several reasons. First, it is a low power device, only consuming 120μA/MHz of active current [87]. Second, it is equipped with 256kB of ferroelectric random access memory(FRAM), which is known for its ultra-low power consumption, high endurance, and fast read/write speeds. Third, it operates in a unified memory model, where SRAM, FRAM, and all the peripherals are mapped in a single global memory, which provides a common interface for all the data that must be secured and checkpointed. Fourth, it contains an on-chip AES accelerator, which can be used to speed-up the cryptographic primitives in SICP.

3.5.2 Three facets

Cause

In our proof-of-concept implementation, the microcontroller is powered by a constant DC power supply. We use a switch to power cycle the microcontroller at arbitrary time intervals to cause power loss.

Statefulness

During the ON-state, the microcontroller stores its general purpose registers, such as program counters (PC), in SRAM and application variables in FRAM. The application variables, found in `.data` and `.bss` sections, are placed in FRAM using the linker description file. After power loss, only FRAM data remains persistent, whereas the SRAM data is lost, leading to memory inconsistency between the volatile and non-volatile program state. We implement a modified version of TI's Computer Through Power Loss(CTPL) utility [96] to maintain a consistent checkpoint across all types of memory. The CTPL utility is designed for TI's `cl430` compiler. It was ported to compile on `msp430-elf-gcc` with changes to preprocessing references and to compile specific assembly code. It was further modified to support user declarable checkpoint functions, to invoke SICP functions within checkpoint calls and to incorporate SICP functions at system startup.

Checkpoint Location and Contents : The memory section containing *STATE* is separately declared as `.checkpoint` section in the device linker file, enabling easy identification of the data to be checkpointed and forcing its location within tamper sensitive memory. It provides a single known location for the `WIPE()` operation to target, discussed in detail later in this section. A guaranteed memory location also allows a straightforward check on

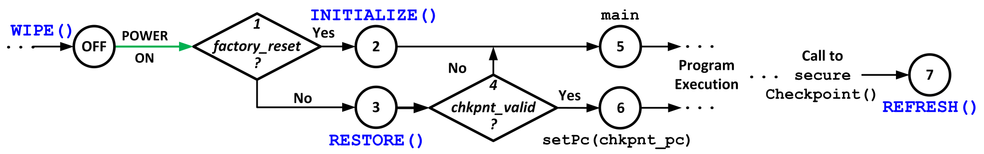


Figure 3.6: Startup sequence for MSP430FR5994 with SICP. (1) Checks for *factory_reset()* and calls (2) `INITIALIZE()` or (3) `RESTORE()` to populate $STATE$ in non-volatile memory. (4) the device inspects $STATE$ for a valid checkpoint, restoring the checkpoint (6) if one is found or invoking `main()` (5) if one does not exist. Program execution will then continue normally until power is lost or another checkpoint is created.

the existence of a *factory_reset()* operation. It provides the application developer a simpler declaration interface, enabling the use of GCC's variable attributes, marked with the `__attribute__` keyword, instead of a complex variable registration interface and tracking data structure. We define `secureCheckpoint()` to generate a checkpoint in this dedicated location and create a state save packet using the SICP algorithms. Figure 3.5 illustrates the control flow involved in creating a checkpoint and subsequent state save packet. First, the volatile peripherals in use are saved on the stack, such as a timer and a comparator. Second, the general purpose registers are pushed on the stack. Since the first two steps mangle the stack and peripheral states, they must be restored to their original state after checkpoint generation. Third, the stack is saved in the `.checkpoint` section. Fourth, the non-volatile data which is to be secured is also stored in the checkpoint along with the volatile state. The checkpoint is ready to be secured by SICP. To create a state save packet, `REFRESH()` is called to wrap the segment up in a valid state save packet.

Security

The four algorithms of the protocol are defined as functions to create and restore the state save packet. We define `REFRESH()` to generate the latest state save packet, `RESTORE()` to restore the latest unmodified state save packet, `INITIALIZE()` to create the first state save packet, and `WIPE()` to wipe sensitive data using the residual power source. `INITIALIZE()` and `RESTORE()` are called automatically during system startup, as shown in Figure 3.6. `WIPE()` is also automatically triggered upon power loss.

3.5.3 System Integration

The modified checkpointing system is wrapped with the SICP function calls to enable secure and stateful power transitions. The device specific implementation of system start-up, cryptographic primitives, and `WIPE()` are as follows:

Startup : Figure 3.6 illustrates the startup sequence for a system employing SICP. A portion of the non-volatile memory region containing $STATE$ is first checked for the factory reset bit pattern. This is used to determine if a *factory_reset()* has occurred and `INITIALIZE()` must be invoked, or a normal boot sequence with `RESTORE()` must occur. In either case, the appropriate SICP function is executed overwriting $STATE$ with either zeros, for `INITIALIZE()`, or the authenticated and decrypted system state, for `RESTORE()`. If the checkpointing system determines that no valid checkpoint exists, such as on the first boot after a *factory_reset()*, it will invoke `main()` as would be expected in a standard system startup.

nonce() : A majority of the nonces used in this protocol are provided via a 128-bit counter that is initialized to zero during `INITIALIZE()` and incremented each time a new nonce is requested. The exception is for the nonce for T_B used in `INITIALIZE()`, which is generated randomly. This nonce is generated randomly to ensure that no two different uses of a device create the same pattern of tags, even if the exact same code is executed following a *factory_reset()* [42, 76].

AEAD Integration : he development of the SICP API is agnostic of the underlying AEAD scheme used to enforce the protocol's security guarantees. We use a hybrid implementation of EAX [15], provided by the Cifra [20] cryptographic library. EAX is a well established two-pass AEAD scheme which avoids unnecessary decryption operations when a tag fails authentication in `REFRESH()` or `RESTORE()`. Tag failure occurs on half of the calls to these two functions since the state save packet authentication is used to determine which packet is valid and which is to be overwritten/restored. The block-cipher based nature of EAX enabled hardware acceleration by modifying the code to employ the MSP430FR5994's AES accelerator.

Tamper free memory : The secure memory is emulated using the Intellectual Property Encapsulation segment (IPE) available in MSP430FR5994 [28]. IPE is used to program a section of FRAM as secure memory, `.secure`, by setting the memory boundaries in the IPE registers. `.secure` section of the memory is programmed with read, write, and execute access. It is used to store the nonce used in SICP and the functions used to read and update the nonce. The variables stored in `.secure` section, and in turn the nonce, can only be read and updated by executing code stored in `.secure` of the memory. The code in `.secure` section can be executed by branching into the IPE segment or by calling a function stored in IPE segment. A read access to `.secure` section from outside the IPE segment will at least return 0x3FFF.

3.5.4 Residual energy use

A residual energy source, which was an architectural requirement, is required to ensure atomicity and to wipe unprotected data after a power loss is detected.

Atomicity Support : The atomicity of `secureCheckpoint()` is ensured by using two state save packet buffers. All changes in non-volatile memory are made to the alternate buffer, such that the most recent packet remains unmodified. Once the new tag computation is complete and stored in a temporary buffer, the `sic_copyTag()` function is called to overwrite the previous tag and set the newly created checkpoint as the only valid checkpoint in an atomic operation. This copy function is made atomic by disabling all interrupts for the copy duration of 48 cycles and relying on the residual energy of the device and the FRAM's atomic byte write capability to ensure that even if power is lost, the copy operation will complete before the system stops operating. `secureCheckpoint()` either has no effect on the system, if power is lost before the tag update, or completes the checkpoint creation without incident.

`WIPE()` : The implementation of the *WIPE* operation requires detection of power loss by monitoring the device's V_{cc}. MSP430FR5994's ADC12_B analog-to-digital converter is used to measure V_{cc} against the system's V_{ref} as described in TI's FRAM Utilities [96]. The MSP430FR5994 development board's unmodified implementation, including one 10μF capacitor and three 100nF capacitors, has sufficient residual energy to consistently overwrite up to 16kB of memory using direct-memory-access (DMA) following the trigger for power loss. When V_{cc} falls below V_{ref} ADC12_B triggers overwrite of *STATE* and SRAM via DMA using the residual energy.

Table 3.2: Executable size overhead of just intermittent computing vs SICP with hardware accelerated EAX and lightweight software only Ketje Sr

Component	**Size (B)**
Checkpoint Support	2532
EAX (HW)	3938
Ketje Sr	3336

3.6 Preliminary Evaluation OF SICP

Results

We demonstrate SICP's feasibility and measure the cost in terms of energy, time and code size overhead incurred to protect a sequence of checkpoints. We have utilized reference implementations for both AEAD designs. EAX(HW), which is a hybrid hardware AEAD primitive, is obtained by substituting the software block cipher operations within EAX with the MSP430FR5994's AES hardware accelerator [4]. The comparison between the performance of the different AEAD schemes is specific to our protocol implementation and is not an evaluation of the different AEAD constructions themselves. All measurements were taken when the microcontroller was operating at 1 MHz and use a state size of 2kB, a reasonable region for applications on a resource constrained device. The energy and time overhead of SICP functions must be measured separately when SS_A and when SS_B are the valid state because the authenticity of SS_A is always checked first in the protocol. The two measurements are then averaged to present the following results.

Overhead Table 3.2 provides an estimate of the expected growth of a program's memory footprint when support for each component is added to the system. EAX(HW) and Ketje SR represent the executable size overhead for SICP functions along with their respective cryptographic kernels. The energy and time overhead are listed in Table 3.3. SICP with

Table 3.3: Energy and time overhead for various algorithms SICP for securing 2kB of checkpoints

Method	`INITIALIZE`		`REFRESH`		`RESTORE`	
	Time (ms)	**Energy (μJ)**	**Time (ms)**	**Energy (μJ)**	**Time (ms)**	**Energy (μJ)**
Checkpoint Support	0.032	0.033	14.4	12.6	14.1	12.8
EAX(HW)	0.061	0.039	355.2	263.2	455.2	332.3
KETJE SR	0.073	0.044	1912.1	15433.3	1301.4	10011.2

EAX(HW) achieves lower overhead compared to KETJE SR because of its two-pass structure and the use of hardware accelerated AES module. In all cases, the overhead incurred by the checkpointing system is constant and is listed under Checkpoint Support in Table 3.2 and 3.3.

Analysis Even though KETJE SR is a lightweight AEAD scheme, it still generates significant overhead within SICP compared to a hardware accelerated version of EAX both in terms of energy and time. This highlights the advantage of hardware accelerated cryptographic modules within SICP. Even with a hardware accelerated AEAD primitive, SICP takes considerable time and energy to secure the checkpointed state, which highlights the need for efficient, lightweight AEAD primitives. The latest advantages in technology scaling does not apply to non-volatile memories. FRAM, one of the most energy efficient non-volatile memories, is only available in 130nm technology. Advances in non-volatile memory technologies will help improve the performance of the protocol.

SICP does not provide any backdoor for the attacker. For example, if an adversary tries to repeatedly cut power to the device during protocol operations, the device continues operation without any glitches because of the atomicity guarantees of the protocol. Similarly, if an adversary tries to emulate a *factory_reset()*, they will be left with a device with a clean memory and newly loaded key. Since a *factory_reset()* wipes all the device memory, any sensitive information the adversary wishes to recover will be unavailable to the attacker.

3.7 Conclusion

We presented the Secure Intermittent Computing Protocol to bridge the gap between stateful power transitions and secure embedded systems. It is the first secure intermittent solution to provide comprehensive security to the power transitions of an embedded system. It is a fail-safe and generic protocol that can be used with existing stateful power transition solutions to enhance their security. We provide a proof-of-concept implementation of secure and stateful power transitions on an MSP430FR5994 to demonstrate the feasibility of secure checkpoints. Several low power microcontrollers are equipped with cryptographic hardware, mostly for encryption. In the future, we must consider including hardware accelerated authenticated encryption engine, low power non-volatile memory, and secure storage capabilities to microcontrollers to improve the duty cycle of the application to facilitate secure and stateful power transitions.

Chapter 4

Optimization Using Configurable Security Levels

4.1 Introduction

Traditionally, IoT devices were powered through a managed power infrastructure, such as a mains connection or a battery. However, this is not scalable; wireline connections prevent IoT devices from becoming truly pervasive, and batteries require periodic replacement. Hence, the rise of IoT devices to truly large scale will go hand in hand with novel ad-hoc power infrastructure in the form of energy harvesting of ambient sources such solar [82], wind [57], RF [82], and vibration [59]. Using a transducer, the ambient energy is converted into electrical energy. The power output from energy harvesters is limited from a few μW to a few W for typical harvesters and therefore has to be accumulated in an energy buffer before the IoT device can be powered up. The use of energy harvesters potentially liberates IoT devices from externally managed energy dependencies.

Although energy harvesters ensure the autonomous operation of IoT devices, they do not guarantee the continuous operation of the IoT device for two reasons. First, the source of ambient energy itself may be discontinuous. Solar cells don't deliver power at night, and vibration energy harvesters don't deliver power when they are at rest. Second, the IoT device itself may consume more power than what can be delivered through energy harvesting. Both of these conditions manifest themselves with the same effect: the energy buffer is depleted and the IoT device needs to power off.

To protect long-running software applications from premature termination through power loss, the IoT device will compute and store a checkpoint in non-volatile memory [78]. The checkpoint will enable the state of the IoT device to be restored after the energy buffer is replenished. The checkpoint includes all the information needed for forward progress including but not limited to microcontroller state, program variables, and peripheral settings. *Intermittent computing* is a collection of techniques that help to create a checkpoint while minimizing the overhead needed to create the checkpoint [62]. The creation and restoration of a checkpoint require energy and clock cycles, which impact the overall performance of the application. Researchers have extensively studied the effects of intermittent power delivery on the application and the IoT device with a primary focus on the efficient and accurate recovery of the application after power loss. Their main focus has been on what to checkpoint, when to create a checkpoint, and how to efficiently generate and restore checkpoints. Typically, they aim to achieve a subset of the following features - continuity of control flow [41, 61, 93, 102], continuity of data flow [44, 61], retention of peripheral state [10, 17, 19, 64], processing time sensitive data [51, 93], and optimizing checkpoint size [8]. While the above features ensure the statefullness of the application, the security of the energy harvested IoT device has been largely ignored.

Motivation Besides the power delivery challenges, IoT devices have to operate correctly and securely. The IoT devices must protect sensitive data either when stored on the device or else when transmitted over the network, they must only accept commands from authorized users, and their operation must be correct and protected from malicious control. Security is not an optional feature; rather it is a fundamental requirement for the promise of IoT to succeed [79]. A broad class of cryptographic algorithms and dedicated security protocols provide the tools and mechanisms to build trust [48]. In addition, security architectures ensure that these cryptographic algorithms themselves operate as expected, free from tampering and malicious influence [31, 63, 70]. However, all of our known cryptographic tools and architectures were created with the basic assumption that power is available and uninterrupted. While there is a trustworthy procedure known as *secure boot* to describe the initialization activities upon power restoration, there is no equivalent set of activities to describe how to create a checkpoint or how to power down a system. Hence, the unique power model of energy harvesting devices presents a novel challenge for our existing solutions to secure architecture.

The challenges of maintaining security across power loss [52], which was often ignored, is an emerging research area in intermittent computing. The security challenges are caused by the intermittent power supply and the non-volatile nature of checkpoints. We broadly classify checkpoint security solutions based on memory isolation [11, 28] and cryptographic primitives [11, 32, 53, 100]. The isolation based techniques use off-the-shelf microcontroller features such as ARM TrustZone and other memory protection units to make the checkpoint inaccessible to the attacker. They use the architectural and hardware properties of the microcontroller to secure checkpoints by controlling access rights to certain memory sections that store the checkpoints. While isolation prevents unauthorized access of checkpoints, only cryptographic primitives encode information security properties, such as confidentiality, integrity, and/or freshness, within checkpoints. In our work, we focus on securing checkpoints using cryptographic primitives [11, 32, 53, 100]. Our work resides at the crossing of intermittent computing and the security challenges required for a secure IoT. In particular, we investigate how application-level security concerns map into the security primitives developed for secure checkpointing. The key contributions of our work are as follows:

- We propose different security levels to configure checkpoint security based on application needs instead of a one-size-fits-all solution.
- We optimize an existing checkpoint security solution based on cryptographic primitives [53] and incorporate the proposed configurable checkpoint security in its implementation.

Organization In the following section, we provide a brief overview of intermittent computing, its security requirements, and the state-of-the-art checkpoint security solutions. In Section 4, we propose a configurable checkpoint security setting that leverages the application to reduce the overhead of securing checkpoints. In Section 5, we describe our implementation of the configurable checkpoint security using a secure checkpoint protocol followed by conclusions in Section 6.

4.2 Background on Intermittent Computing

We briefly provide a background on the minimum security requirements of checkpoints and their design in state-of-the-art checkpoint security solutions. We introduce intermittent computing and its security properties using Cyclic Redundancy Check (CRC) as an example intermittent application. CRC is widely used in several protocols, such as BLE [99] and IEE 802.15.4 [43], to detect erroneous input data. We consider a microcontroller powered by an energy harvester which operates in an intermittent computing model, as illustrated in Figure 4.1. The microcontroller receives the input data, `CRCInput`, and its expected 32-bit code, `CRC32Expected`, which is verified by `CRC32Calculate()` function.

When the microcontroller loses power before CRC verification, it creates a checkpoint of the necessary state required for the forward progress of the application using *refresh* operation.

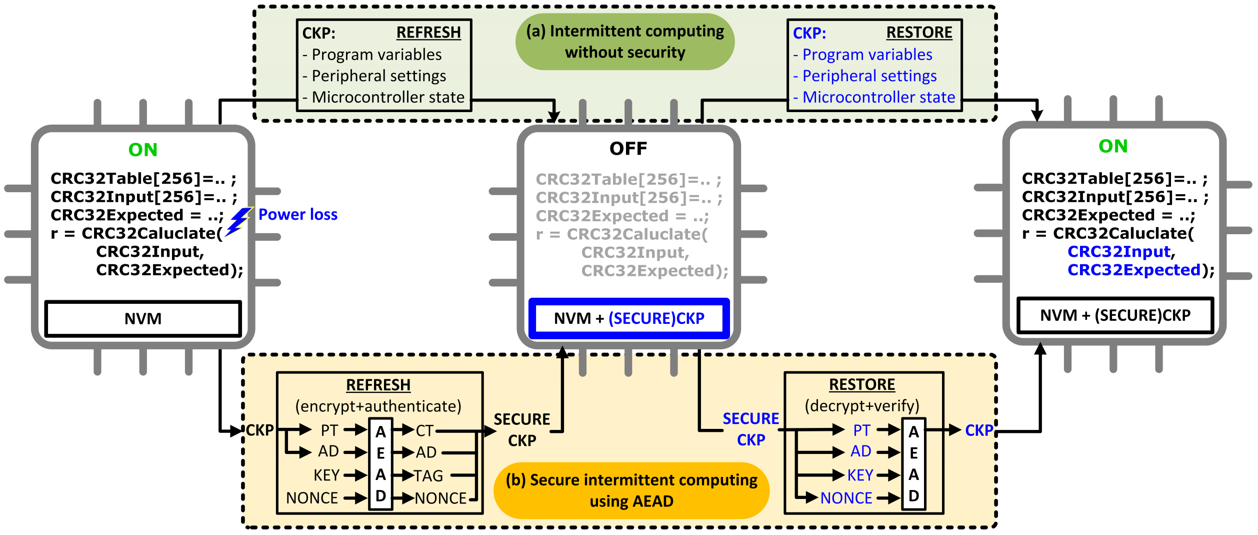


Figure 4.1: CRC32 verification as an example intermittent application running on a microcontroller with non-volatile memory(NVM). (a) Unsecure intermittent computing that stores plaintext checkpoints(CKP) and restores checkpoints without any security checks. (b) Secure intermittent computing using AEAD to encode security properties such integrity, authenticity, confidentiality, and freshness into secure checkpoints (SECURE CKP) which are verified before restoring the decoded checkpoint (CKP).

In the top half of Figure 4.1, the checkpoint (CKP) contains program variables, peripheral settings, and microcontroller state. We elaborate on the contents of a checkpoint in Section 5.3.3. When there is sufficient harvested energy, the microcontroller is powered-on again and the checkpointed state is *restored* from non-volatile memory (NVM). The CRC verification resumes with the checkpointed input and is completed, provided the input power supply is not interrupted. The number of checkpoints required to complete this CRC verification depends on the frequency of power losses, where a new checkpoint is generated with every power loss. We identify forward progress as a minimum requirement for the meaningful and practical application of intermittent computing. In between two power-loss events, there should be enough energy available to restore a checkpoint, to execute at least one instruction of the CRC application, and to re-save the latest progress in a new checkpoint. If this requirement is not met, then the intermittent computing scenario is not able to make *forward progress* in the application; the entire available energy budget is used to save and restore checkpoints.

4.2.1 Security In Intermittent Computing

In intermittent computing without security, the microcontroller creates a checkpoint when needed and restores the most recent valid checkpoint, as illustrated in the top half of Figure 4.1. While this is a stateful computation model, it does not guarantee the statefullness of security properties. The checkpoint may be tampered in non-volatile memory and the microcontroller will restore to a malicious state when using the tampered checkpoint. If the attacker can read from and write to non-volatile memory, they can snoop, spoof, and replay checkpoints [52]. Unsecure intermittent computing not only introduces vulnerabilities to the application using checkpoints, it also weakens the security architectures and algorithms used to secure the application.

Checkpoint security requirement: At a minimum, intermittent computing must ensure the statefulness of a few security properties along with the forward progress of the application. First, the checkpoint integrity and authenticity must be protected to prevent unauthorized modifications to the checkpoint and to ensure that checkpoints cannot be replayed on an attacker controlled device, respectively. Second, the freshness of the checkpoint must be guaranteed to prevent the replay of a stale checkpoint on the same microcontroller which may affect the control flow of the application. Third, the availability of a valid checkpoint must always be guaranteed to ensure the microcontroller does not restart the application because of the lack of a valid checkpoint. Finally, the checkpoint may require confidentiality guarantees based on the contents that require protection from unauthorized access.

Checkpoint security properties	Ghodsi et al [32]	SECCS [100]	Asad et al [11]	SICP [53]
Integrity & Authenticity	–	✓	✓	✓
Freshness	–	–	–	✓
Availability	–	–	–	✓
Confidentiality	✓	✓	✓	✓

Table 4.1: Checkpoint security properties satisfied by the state-of-the-art related work

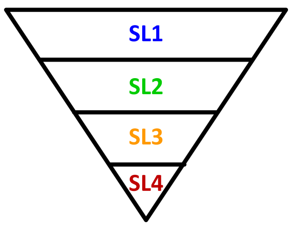
Related work: Table 4.1 lists a few state-of-the art solutions for secure intermittent computing that satisfy a subset of the above security requirements. Ghodsi et al. [32] only encrypt the checkpoint without considering the other requirements. SECure Context Saving (SECCS) [100] and Asad et al. [11] ensure the information security of checkpoint, including confidentiality, integrity, and authenticity, using encryption and authentication algorithms such as Authenticated Encryption with Associated Data(AEAD) [81]. The Secure Intermittent Computing Protocol (SICP) [53] satisfies all the minimum security requirements. SICP also uses AEAD to ensure the information security properties of the checkpoint and its freshness. SICP is the only solution that ensures the availability of checkpoints by always storing two checkpoints, i.e., the latest and the previous checkpoint.

Common cryptographic primitive for securing checkpoints: A majority of cryptographic checkpoint security solutions [11, 53, 100] use AEAD or a combination of encryption and authentication to protect checkpoints. We capitalize on the versatility of AEAD in Section 4.3 to implement configurable checkpoint security. Here, we explain how the security properties are encoded into the checkpoints with AEAD using the bottom half of Figure 4.1. AEAD uses a secret key (authenticity) and a unique nonce (freshness) to encrypt and authenticate checkpoints. AEAD also takes in associated data as input which is plaintext information that only needs integrity and authenticity, but not confidentiality. The refresh operation encrypts (confidentiality) and authenticates (integrity) the checkpoint using AEAD to generate ciphertext, and to generate authentication tag over ciphertext and associated data if provided. The ciphertext, authentication tag, associated data, and nonce are stored in non-volatile memory as a secure checkpoint. The restore operation verifies the authenticity and integrity of the ciphertext, associated data, and authentication tag using AEAD before restoring the microcontroller with the decrypted checkpoint.

4.3 Configurable Mulli-level Checkpoint Security

A checkpoint contains a snapshot of all the data necessary to resume the progress of the application. As described in the previous section, the contents of the checkpoint are largely dependent on the application. Let us consider CRC32-HW benchmark. Apart from device specific data such as the stack and general purpose registers, the checkpoint also contains the incoming data frame and the registers of the CRC peripheral in program variables. The existing checkpoint security solutions incorporates a single security policy to the entire checkpoint. For example, if a programmer decides to use SECCS [100] to secure their checkpoints, then the entire checkpoint will be encrypted and authenticated. Similar to SECCS, the other solutions listed in Table 4.1 follow the same one-size-fits-all policy to secure its checkpoints. Even if the application does not require encryption of the entire checkpoint, the programmer ends up encrypting the entire checkpoint because of the nature of existing checkpoint security solutions. This is detrimental to the forward progress of the application as the encryption consumes a portion of the harvested energy which may otherwise be used by the application.

By being more selective in deciding what parts of the program state and device specific state should be encrypted, considerable performance trade-offs can be made. We propose four security levels (SL) for checkpoints based on a combination of the security requirements



Security Properties				Overhead	Checkpoint division	
Confidentiality	Integrity & Authenticity	Freshness	Availability		Plaintext	Associated data
●	●	●	●	++++	●	-
◗	●	●	●	+++	◗	◗
-	●	●	●	++	-	●
-	-	-	-	+	-	-

Figure 4.2: Proposed levels for checkpoint security with an decreasing overhead for securing checkpoint and decreasing guarantees for security properties from SL1 to SL4. The decreasing overhead corresponds to the decreasing size of plaintext input in checkpoint partition, where larger plaintext input to AEAD increases overhead from encryption and decryption.

provided by the state-of-the-art in checkpoint security. In this section, we demonstrate how to achieve the generic optimizations involved in multi-level checkpoint security using a select solution from Table 4.1. We also propose certain optimizations specific to the selected solution to minimize the overhead from securing checkpoints.

4.3.1 Multi-level Checkpoint Security

Our multi-level checkpoint security involves four levels, illustrated in Figure 4.2. We leverage the design of AEAD described in Section 4.2 to realize the security properties in each level. The security properties of SL(i) are a subset of the security properties of SL(i+1).

SL4: No security

With the least overhead incurred, SL4 does not guarantee any security properties for the checkpoints of an intermittent system. It incurs the least overhead as no cryptography is involved in the encoding of a checkpoint. SL4 is equivalent to unsecured intermittent computing systems.

SL3: No confidentiality

If an application does not generate checkpoints with sensitive content that require confidentiality, SL3 is sufficient to ensure the forward progress of the application's security features. We propose checkpoint integrity, authenticity, freshness, and availability as the minimum requirement in checkpoint security irrespective of the contents of checkpoints, which is satisfied by using SL3. These three requirements are guaranteed for any associated data input to the AEAD algorithm. With AEAD, we consider the entire checkpoint to be associated data and with no plaintext as the checkpoint in SL3 does not require confidentiality guarantees.

SL2: Partial confidentiality

A few applications may contain sensitive data of long running applications such as key exchange, for which it suffices to only encrypt sensitive sections of checkpoint while maintaining the SL3 properties for the rest of the checkpoint and encrypted sensitive data. We achieve SL2 by partitioning the checkpoint into public and private sections. For example, we may consider all the program variables in Table 5.3 to be private and the device specific state to be public. Both the public and private sections require SL3 level security guarantees, whereas, the private section also requires confidentiality guarantees. The private section is input as plaintext to AEAD and the public section is used as associated data.

SL1: Full confidentiality

SL1 guarantees the confidentiality of the entire checkpoint and guarantees SL3 properties for the encrypted checkpoint. The security properties of SL1 are also guaranteed using AEAD, by using the entire checkpoint as plaintext data. SL1 provides a comprehensive solution to secure checkpoints, and at the same time, provide us with a base metric to compare the advantage of SL2 and SL3 over SL1, which is similar to the state-of-the-art solutions in Table 4.1 that employ one-size-fits-all security policy to the entire checkpoint.

4.3.2 Configuring And Optimizing Checkpoint Security Using SICP

Selecting A Checkpoint Security Solution

We chose the Secure Intermittent Computing Protocol (SICP) [53], described in Chapter 3, to demonstrate multi-level checkpoint security for three reasons. First, it satisfies all the minimum security guarantees required for protecting the checkpoints of an intermittent system, which ensures that our multi-level secure intermittent computing is incorporated into SICP without modifying the original cryptographic protocol. In particular, it is the only solution in Table 4.1 that ensures the availability of a secure checkpoint which is important as the threat of power loss is imminent in intermittent systems. Second, it is a generic software solution that can be easily adapted to any intermittent computing technique, which helps demonstrate that multi-level checkpoint security is also accessible to any intermittent computing technique. We demonstrate this advantage using an implementation on a commercial off-the-shelf device in the next section. Third, SICP also uses an AEAD scheme at the core to achieve its security properties which easily guarantees selected security properties for different sections of the checkpoint, as discussed in Section 4.3.1.

SICP Review

We provide a brief overview of the protocol to help understand the techniques used to implement multi-level security and the protocol specific optimizations proposed below. The freshness requirement is guaranteed using a 128-bit nonce, R, associated with each checkpoint which is passed onto AEAD as an input. The information security requirements are

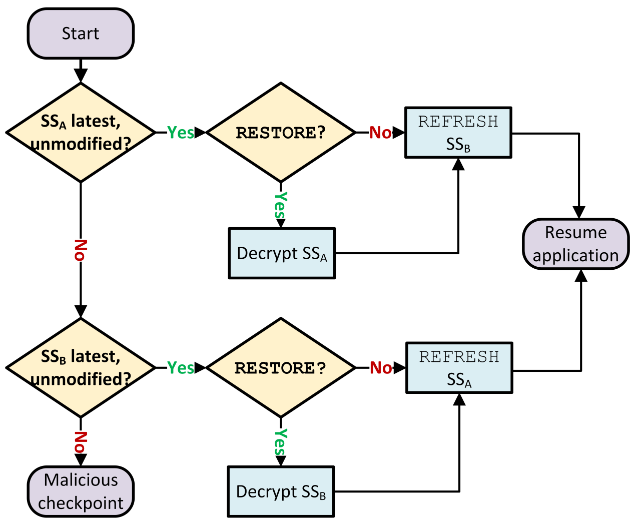


Figure 4.3: A flow chart of the original SICP algorithm `REFRESH` and `RESTORE` using state save packet, SS_i , to store secure checkpoints in alternating buffers A and B. Both the algorithms detect the latest unmodified buffer cryptographically using AEAD. `RESTORE` creates a new state save packet with a new nonce and the latest checkpoint without any forward progress in the application.

guaranteed by using the checkpoint as plaintext input to AEAD encryption to generate encrypted checkpoint and authenticated tag, T. The nonce and the secret key used by AEAD are stored in tamper-free non-volatile memory, which is protected from malicious access. One may argue that placing the entire checkpoint in tamper-free memory may prevent the attacker from tampering with checkpoints. While this may be a potential checkpoint security solution, it is not applicable for all benchmarks and devices. The size of the checkpoint varies based on the benchmark, as listed in Table 5.3, and the size of tamper-free memory is dependent on the platform. SICP uses a two-state secure checkpoint buffer, A and B, and updates them alternatively to maintain availability guarantees. The authentication tag from the previous checkpoint is used as associated data input for the latest checkpoint to ensure only one of the buffers contains a valid checkpoint. Figure 4.3 illustrates the flow of generation and restoration of a secure checkpoint, also known as state save packet, SS, with encoded security properties which contain the encrypted checkpoint, authentication

	AEAD mapping				Section mapping			
	SL4	**SL3**	**SL2**	**SL1**	**SL4**	**SL3**	**SL2**	**SL1**
Program Variables	-	AD	AD+P	P	-	Pub	Pub+Pri	Pri
Device Specific State	-	AD	AD	P	-	Pub	Pub	Pri

Table 4.2: Mapping of checkpoint partitions, program variables and device specific state, as inputs to AEAD and as different memory sections at the proposed levels of security. The plaintext (P) input is the `.private` (*Pri*) section of memory and the associated data (AD) input is the `.public`(*Pub*) section of memory. The size of each section of memory depends on the size of checkpoint partitions. SL4 does not require partitioning of checkpoint as there are no security properties. In SL2, the program variables are divided between *Pri* and *Pub* to apply confidentiality properties to S section.

tag, and the nonce. Both `REFRESH` and `RESTORE` identifies the latest checkpoint between A and B by first verifying the integrity and authenticity of checkpoint A. If buffer A is the latest unmodified checkpoint, `REFRESH` directly generates a new checkpoint in buffer B and `RESTORE` decrypts checkpoint A before generating a new checkpoint in buffer B with the restored state and a new nonce. If buffer A is not the latest unmodified checkpoint, buffer B is checked for the same, and buffer A is updated.

Algorithm 3 defines the two most important SICP primitives, `REFRESH` and `RESTORE`. The encryption and decryption operations of AEAD are divided into $AEAD_{encr} + AEAD_{auth}$ and $AEAD_{decr} + AEAD_{auth}$ for clarity. SICP always stores the latest and the previous checkpoint to ensure the availability of a secure checkpoint even if the latest checkpoint is incomplete. Apart from `REFRESH` and `RESTORE` described in Algorithm 3, SICP also performs `INITIALIZE`, which creates the first secure checkpoint, and `WIPE`, which is automatically triggered upon power loss to erase secure sensitive sections of volatile and non-volatile memory. We refer readers to the detailed implementation of these protocol steps [53].

Partitioning Checkpoints For Multi-level Security

SICP originally ensured freshness, authenticity, integrity, and confidentiality of the entire checkpoint. We identified the contents of checkpoints and defined their security properties to incorporate different security properties for each part of the checkpoint. Since the contents are specific to an application, we assume the programmer defines the security requirements for the contents of the checkpoint. If the programmer chooses either SL1 or SL3, they will apply the same security properties across the entire checkpoint. Whereas selecting SL2 involves partitioning the checkpoint into public, *Pri*, and public sections, *Pub*, as described under section mapping in Table 4.2.

Now, the checkpoint is divided into a public section, *Pub*, which requires integrity, authenticity, and freshness, and a private section, *Pri*, which additionally requires confidentiality guarantees. By design, SICP uses AEAD to secure checkpoints. In SICP, the plaintext was the entire checkpoint and associated data was just the authentication tag from previous checkpoints. With configurable checkpoint security, the plaintext provided to AEAD is only the private section of the checkpoint. The rest of the checkpoint, which is in the public section, is provided to AEAD as associated data along with the authentication tag from the previous checkpoint, as in the original SICP. Table 4.2 states the one-to-one mapping between associated data and public section, and plaintext and private section.

After partitioning the checkpoint, to achieve each of the different security levels, the programmer needs to modify the following inputs to AEAD in the original SICP, as illustrated in Algorithm 3. Table 4.2 uses the broad partition of checkpoints to map the contents to security properties using AEAD inputs and memory sections for each level of security. SL4 does not require partition of checkpoint or use of AEAD as there are no security properties encoded at this level. In SL3, since no part of the checkpoint is encrypted, the entire check-

Algorithm 3 Optimized SICP: `REFRESH` and `RESTORE`

```
Require: $K, STATE, S_i, NS_i, R_i, T_i, f_i$ where $i \in \{A, B\}$
        $operation \in \{$REFRESH, RESTORE$\}$
 1: $Q \leftarrow nonce()$
 2: if $f_A = 1$ and $f_B = 0$ then
 3:     if $operation$ = RESTORE then
 4:         if $T_A = AEAD_{auth}(S_A, T_B|NS_A, R_A, K)$ then
 5:             $STATE \leftarrow AEAD_{decr}(S_A, T_A, T_B|NS_A, R_A, K)$
 6:         end if
 7:     end if
 8:     if $operation$ = REFRESH then
 9:         $R_B \leftarrow Q$
10:         $S_B \leftarrow AEAD_{encr}(STATE, T_A|NS_B, R_B, K)$
11:         $T_B \leftarrow AEAD_{auth}(S_B, T_A|NS_B, R_B, K)$
12:         $f_A, f_B \leftarrow (0, 1)$
13:     end if
14: else if $f_B = 1$ and $f_A = 0$ then
15:     if $operation$ = RESTORE then
16:         if $T_B = AEAD_{auth}(S_B, T_A|NS_B, R_B, K)$ then
17:             $STATE \leftarrow AEAD_{decr}(S_B, T_B, T_A|NS_B, R_B, K)$
18:         end if
19:     end if
20:     if $operation$ = REFRESH then
21:         $R_A \leftarrow Q$
22:         $S_A \leftarrow AEAD_{encr}(STATE, T_B|NS_A, R_A, K)$
23:         $T_A \leftarrow AEAD_{auth}(S_A, T_B|NS_A, R_A, K)$
24:         $f_B, f_A \leftarrow (0, 1)$
25:     end if
26: else
27:     $abort()$
28: end if
```

point is considered public and passed as associated data. In SL1, the entire checkpoint is in *Pri* and provided as plaintext input to AEAD as in the original SICP. In SL2, *Pri* and *Pub* are inputs for plaintext and associated data, respectively, in AEAD operations.

SICP Optimizations

We studied SICP design to reduce overhead from the security operations to perform design specific optimizations. We propose two optimizations that avoid unnecessary encryption/authentication (OPT1) and decryption/verification (OPT2) operations, illustrated in Algorithm 3.

OPT1 Avoid re-encrypting the checkpoint in `RESTORE`: SICP creates a new secure checkpoint upon every power up to keep track of the number of power cycles using the nonce (counter), which creates a new secure checkpoint without any forward progress in the application. We propose not to re-encrypt the checkpoint after restoring the microcontroller with the latest checkpoint. While re-encryption may be useful for certain applications, it consumes extra energy and time for securing a checkpoint without any progress in the application. With our optimizations, we resume forward progress of the application after verification and restoring the decrypted checkpoint.

OPT2 Identify the latest checkpoint using a 1-bit flag: Both `RESTORE` and `REFRESH` originally decrypted/verified one of the checkpoints first to identify the latest checkpoint, which was either restored or left unchanged to update the other checkpoint buffer, respectively. This verification failed half the time because checkpoint A was always checked for newness before checkpoint B. We propose to avoid this failed cryptographic verification step by using a single-bit flag to indicate newness. f_A is set and f_B is reset to indicate A is the latest checkpoint and vice versa, as listed in lines 12 and 24 in Algorithm 3. The flags are stored in the secure non-volatile memory to prevent the attacker from invalidating both the checkpoints and triggering unnecessary decryption/verification. The optimized SICP always checks for the secure checkpoint with a set flag to either restore the decrypted checkpoint if it passes the verification check or updates the other checkpoint buffer with the latest checkpoint. This flag check is added on lines 3 and 15 in Algorithm 3.

4.4 Implementation

In this section, we present a detailed overview of implementing our proposed configurable multi-level checkpoint security using SICP. We utilize MSP430FR5994, described in Section 5.3.1, to present the details of selecting an AEAD primitive used to secure checkpoints, implementing SICP optimizations, incorporating multi-level security in SICP, and evaluating our implementation.

AEAD Primitive	Energy (uJ)	Time (ms)
EAX (AES-HW)	23.4	9.3
Ascon	90.9	33.9
`GIFT-COFB`	633.6	233.6

Table 4.3: Selecting an AEAD primitive among three ciphers - EAX implemented using on-chip hardware accelerator, 16-bit optimized ASCON implementation, and reference implementation of `GIFT-COFB`. The overhead listed is measured for encrypting and authenticating 16B each of plaintext and associated data using 16B key and nonce.

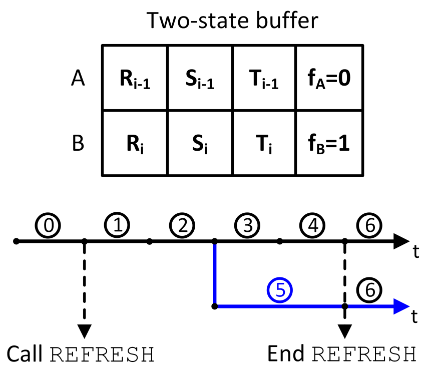
4.4.1 Cryptographic Primitive

We evaluated the performances of several AEAD schemes on MSP430FR5994 to chose the least energy hungry primitive for securing checkpoints. First, we evaluated the finalists from NIST LWC competition[69]. In Table 4.3, we present performance overhead of two selected ciphers, ASCON [29] and `GIFT-COFB` [12]. ASCON was the only cipher with 16-bit optimized submission which was suitable for our 16-bit target platform. We present the performance overhead of 16-bit optimized ASCON as a representative of optimized software implementations of a lightweight cryptographic scheme. `GIFT-COFB` was chosen as a representative of the rest of the submissions with reference, 32-bit optimized or 64-bit optimized implementations. We present the results of the reference implementation provided with the `GIFT-COFB` submission. Next, we also selected EAX [15] as a representative of hardware accelerated AEAD schemes. Our target device is equipped with AES256 accelerator for encryption and decryption.

Table 4.3 provides the energy and time required to encrypt/authenticate fixed inputs across the three selected ciphers. The overhead presented includes AEAD encryption operation for each cipher processing 16B of plaintext and 16B of associated data using a 16B key and nonce to generate 16B of ciphertext and 16B of the authentication tag. A similar overhead was observed for decryption and verification. We are not comparing EAX, ASCON, and `GIFT-COFB` in our experiments, rather, we are evaluating the performance of a hardware accelerated cipher, target architecture optimized software implementation of a cipher, and a reference implementation of a cipher. Our target architecture and application are both resource hungry, thus it was imperative to choose an AEAD scheme with minimal overhead in both energy and time. As expected, from Table 4.3, hardware accelerated EAX consumes the least amount of energy and time. We conclude that when EAX (HW-AES) is used as AEAD cipher to secure checkpoints of benchmark applications, it will consume less harvested energy for securing checkpoints when compared to optimized and referenced implementations of Ascon and `GIFT-COFB`, respectively.

4.4.2 Implementing Optimized SICP

SICP was originally implemented as a library on top of CTPL. We utilize the same approach and add optimization to SICP library. We modified CTPL to add user defined SICP functions that can be called to initialize the protocol, generate secure checkpoints, wipe secure state, and restore unmodified secure checkpoints. SICP uses a 128-bit counter initialized to a random number as the nonce for maintaining checkpoint freshness. SICP collects the checkpoint data provided by CTPL and the nonce, processes them using hardware accelerated EAX to encode the security properties into the secure checkpoint, and stores the output (which is the secure checkpoint) in non-volatile memory. Upon power loss, SICP zeroises all memory sections containing sensitive plaintext data to prevent unauthorized access [39]. In our implementation, we emulate tamper-free memory using Intellectual Property Encapsulation (IPE) feature provided by TI. Upon power-up, the latest checkpoint is verified and decrypted using the secret key from tamper-free memory and the benchmark resumes execution.



Microcontroller Operation	Valid buffer on power loss during operation
⓪ Benchmark	B
① Write $R_A=R_{i+1}$	B
② Write $S_A=S_{i+1}$	B
③ Write $T_A=T_{i+1}$	NONE
④ Write $f_{A,B}=(1,0)$	NONE
⑤ Atomic write T_A=Ti+1, $f_{A,B}=(1,0)$	A
⑥ OFF	A

Figure 4.4: Availability of a valid checkpoint when atomic vs non-atomic memory write operations are used to update authentication tag and flags using residual capacitance. The microcontroller performs a series of operations during a time interval. (0) executes benchmark at the end of which `REFRESH` is triggered, identifies B as the latest buffer, and updates A .(1) Write nonce, (2) Write encrypted checkpoint, (3) Write authentication tag, (4) Write flags, and (6) microcontroller is idle until powered off. The write operations in 1-4 are non-atomic and power loss during (3) and (4) leaves the microcontroller without a valid buffer. Alternatively, atomically writing tag and flags using (5) ensures the write operation is completed and A is the latest buffer even if the device loses power.

Atomic operations Since our input source is intermittent, we must ensure that certain memory writes are performed atomically. Figure 4.4 illustrates the need for atomic update of the authentication tag and flags in step 11, 12 and 23, 24. We disable all interrupts during these writes, to ensure the write operations are completed using the residual capacitors even if the microcontroller experiences a power loss. The atomic write implementation ensures the availability feature provided by the two-state checkpoint buffer in SICP is implemented correctly. A single secure checkpoint buffer might satisfy the availability requirement if the residual on-chip capacitors provided sufficient energy to write checkpoints of varying sizes. But the size and availability of residual capacitance are platform dependent. For example, MSP430FR5994 LaunchPad Development Kit contains one 10μF capacitor and three 100nF capacitors which provide sufficient residual energy to consistently overwrite up to 16kB of memory after detecting a power loss. To provide a generic implementation, we use a two-state secure checkpoint buffer and implement the availability feature with atomic writes using residual capacitance. At a minimum, the residual capacitance must provide sufficient energy to power-up the device to finish writing 130-bits of data to non-volatile memory, i.e., 128-bits of authentication tag and 2-bits of flags. We also use the residual capacitance to zeroise unencrypted sensitive sections of checkpoints to prevent unauthorized access.

Optimizations OPT1 is simple, it avoids re-encryption after restoring. In our implementation, we resume the benchmark after the latest, verified benchmark is restored by the microcontroller. OPT2 adds a 1-bit flag to identify the latest checkpoint. We associate a 1-bit tag with each checkpoint buffer A and B, apart from the nonce and the checkpoint itself. This tag is updated atomically to ensure at all times, only one flag is set between f_A and f_B.

4.4.3 Multi-level Checkpoint Security

We use MSP430FR5994's linker description file to define two new sections of non-volatile memory. First, we define the private section, `.private`, from 0x10000 to 0x10FFF. Second, we define the public section, `.public`, from 0x11000 to 0x11FFF. We chose 4kB for each section but the size can be varied depending on the application needs. Among our selected benchmarks, the floating point benchmark generated the largest checkpoint with 3198 bytes, which fits in 4kB of secure or non-secure memory. With well-defined memory sections, the programmer has control of the location of checkpoints which in turn controls the security properties of the contents. We use `__attribute__((section (".private")))` to place sensitive checkpoint data in private memory, as illustrated using the following code snippet from floating point benchmark. All the other checkpoint data is placed in public memory using `__attribute__((section (".public")))`. For ease of use, we define preprocessor directives, such as `PUBLIC_BENCH` to place each part of the checkpoint, including the program variables, microcontroller data, and peripheral data, in either private or public sections.

```
#ifdef PUBLIC_BENCH
    __attribute__ ((section (".public")))
#else
    __attribute__ ((section (".private")))
#endif
float output[15][15];
```

During `REFRESH`, we first check for the size of used private and public sections which provides the size of plaintext and associated data provided to AEAD. We then provide the plaintext (Pri) and associated data ($PubS$) as input to $AEAD_{encr}()$ and $AEAD_{auth}()$. The size of each memory section varies depending on the security level and the size of the checkpoint, as described in Table 4.2. From Table 5.3, we can see the varying checkpoint memory requirement of each benchmark. In all our implementations, SL1 and SL3 contains the entire checkpoint in either public or a private section. We place the peripheral registers as a part of device specific state in public memory in SL2.

4.5 Quantizing Checkpoint Security Levels

In this Chapter, we provide a preliminary estimation of performance improvements achieved using the optimizations proposed in this dissertation. We use the CRC32 example described in Section 4.2 to demonstrate the advantages of selecting checkpoint security requirements

Table 4.4: Performance improvements achieved using configurable security levels, OPT1 and OPT2 when compared to unpotimized SICP (last column)

	SL4		**SL3**		**SL2**		**SL1**		**SICP**	
	E (uJ)	**T (ms)**	**E (uJ)**	**T (ms)**	**E (uJ)**	**T (ms)**	**E (uJ)**	**T (ms)**	**E (uJ)**	**T (ms)**
CRC32	2.2	0.7	202.3	72.5	642.7	240.0	727.8	274.1	1580.4	605.5

E: Energy, T: Time

based on the needs of the application. In our experiment, we compute and verify CRC32 code using a software implementation [72]. The net checkpoint size for CRC32 is 1976B.. The program state consumes 78% (1544B) of the checkpoint and the device specific state occupies the rest (432B) of the checkpoint. If we use the unoptimized SICP, without configuring security levels, OPT1, and OPT2 optimizations, secure intermittent implementation of CRC32 would encrypt and authenticate the entire checkpoint. Whereas, if CRC32 does not require confidentiality then we can avoid the overhead of encrypting 1976B by selecting SL3. But if CRC32 program variables require confidentiality, then we can avoid the overhead of encrypting 432B by selecting SL2.

Table 4.4 lists the energy and time overhead of securing the checkpoints of CRC32 software implementation at different levels of security. We compare the overhead incurred from securing checkpoints at each level of security and unoptimized SICP presented in Chapter 3. The results presented in this section were measured on an MSP430FR5994 LaunchPad Development kit from Texas Instruments [97]. The measurements were collected using Digilent Analog Discovery 2 USB Oscilloscope across a 1 $k\Omega$ shunt resistor. The LaunchPad was operated at 8MHz and powered using an external DC power supply of 3.5V.

The measurements listed under SL1 and SICP encrypt and authenticate the entire checkpoint but SL1 utilizes OPT1 and OPT2 described in this chapter. OPT1 avoids re-encrypting the checkpoint after restoring which saves encryption overhead from 1976B of checkpoint data. OPT2 avoids unnecessary decryption and verification before generating a new checkpoint using flags to indicate the latest checkpoint. SL1, with OPT and OP2, performs approximately 50% better than unoptimized SICP. SL3 consumes approximately 72% less energy and time when compared to SL1 to provide the necessary checkpoint security if CRC32 does not require any checkpoint confidentiality. Similarly, SL2 consumes 12% less energy and time when compared to SL1 by not encrypting non-confidential sections of the checkpoint.

4.6 Conclusion

We proposed a configurable checkpoint security solution based on four different levels of security, SL4 to SL1, which leverages AEAD to customize checkpoint security needs of the benchmarks. We partitioned the checkpoints into public and private sections to avoid unnecessary encryption/decryption of the public section of the checkpoint while ensuring other checkpoint security properties are still full-filled. We provide a proof-of-concept implementation of our multi-level security using a secure checkpointing protocol on a low power microcontroller. In this work, we performed a coarse grained partition of checkpoints based on two broad types of checkpoint content - program variables and device specific state, where all details about the partition are provided by the programmer. In the future, we plan to delve further into the contents of checkpoints to perform fine-grain analysis of checkpoint security properties and automate the partition with minimum input from the programmer using a compiler.

Chapter 5

Benchmarking Secure Intermittent Systems

5.1 Introduction

Unlike conventional computing with near unlimited input power, intermittent computing operates on a limited input power budget that needs to be used judiciously towards forward progress of the intermittent application. While security is not free, it is important to intermittent systems, as described in Chapter 2 and Chapter 3. There are several components in intermittent systems that consume the energy harvested from ambient sources such as the embedded device, intermittent application, hardware peripherals, non-volatile memory, intermittent computing, and security. The amount of input energy consumed by each of the above components determines the energy available for forward progress of the intermittent application.

First, the energy consumed by the embedded device or microcontroller to operate its CPU is dependent on architecture and ultra low power design. Second, the intermittent application, which is programmed on the embedded device, consumes either a small amount or a majority of the input energy depending on the type of application. Third, the intermittent application may utilize hardware accelerated peripherals to reduce the net energy used by the application and to speed-up its computation. Fourth, the type of non-volatile memory used by intermittent systems determines the energy used for reading from and writing to non-volatile memory. Fifth, the type of intermittent computing technique used determines the amount of energy used for generation and restoration of checkpoints to complete the intermittent application. And finally, the type of checkpoint security employed by intermittent systems also determines the overhead incurred in securing checkpoints which limits energy available for forward progress of the application. In this chapter, we primarily study the two-way dependency between checkpoint security and intermittent application.

On the one hand, we study the effect of adding checkpoint security to forward progress of the application. We investigate the effect of intermittent computing and checkpoint security on a real-life application. We selected Elliptic Curve Diffie Hellman Key Exchange (ECDH) as the intermittent application. We use duty cycle of the load as a metric to evaluate the effects of secure intermittent computing on the application's forward progress. We also investigate the effect of other factors such as non-volatile memory on the duty cycle of the application.

On the other hand, we study the role of application in determining the checkpoint security properties. We investigate how applications impact the security requirement which in turn affects forward progress of the applications. Because securing a checkpoint requires energy and time (clock cycles), less harvested energy remains for the application. Hence, secure checkpoints will further reduce the performance of the application. We aim to quantify the impact of security on the overhead of intermittent computing applications. It is important to perform this cost analysis on applications for two reasons. First, the checkpoint size is determined by the application. It is a major factor in analysing the overhead of secure checkpoints. Second, the contents of the checkpoints are also dependent on the application. This determines the security properties required for checkpoints. We make the following contributions in this Chapter:

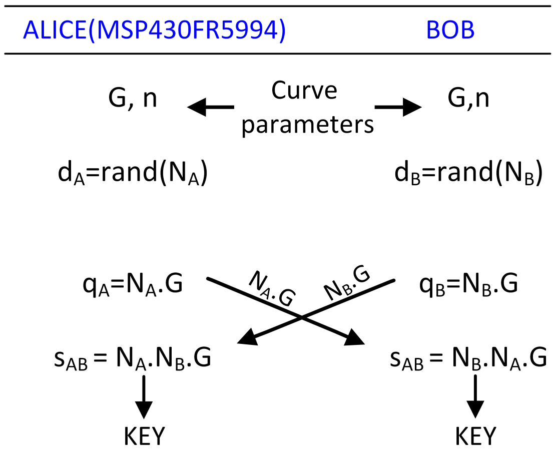
- We study the change in duty cycle based of ECDH in a secure intermittent system.
- We analyze factors such checkpoint size, non-volatile memory, and cryptographic primitives that also affect application duty cycle.
- We analyse the role of application in the overhead of intermittent computing and its security using a curated list of IoT benchmark applications.

5.2 Effect of Checkpoint Security on Application

In this section we evaluate SICP based on a real-life application. We introduce our performance metric, our target application, our implementation of the application, its checkpoints and the need for checkpoint security, and the effect of each facet on the application.

5.2.1 Duty Cycle

Equation 5.1 states the relation between the load duty cycle, D_{load}, the average power available from the harvester, P_{EH}, and the net power required by the load P_{load}. If the load was supplied by a constant power supply, the supply and demand will match, i.e., P_{EH} would equal P_{load}, in which case the device would operate at 100% duty cycle. The harvested power typically does not match the power required by the load. For example, consider a load which performs a cryptographic signature [71] powered by a kinetic energy harvester [25]. Each signature requires 7.3 mW (P_{load}), whereas the harvester only supplies an average of 2 mW (P_{EH}). The load can only operate at a 27% duty cycle to compute signatures. Thus, the duty cycle is determined by the power budget available from the energy harvester.



```
ECDH PSEUDO CODE:
receive(q_B);
ec_curve_gen_(G);
ec_curve_get_ord(n);
bn_rand_mod(d_A,n);
secureCheckpoint();
ec_mul(q_A,G,d_A);
secureCheckpoint();
send(q_A);
ec_mul(s_AB,q_B,d_A);
secureCheckpoint();
kdf(k_AB, s_AB);
```

Figure 5.1: ECDH key-exchange between Alice(MSP430FR5994) and Bob: a flow chart and pseudo-code.

$$D_{Load} = \frac{P_{EH}}{P_{Load}} \times 100 \tag{5.1}$$

Since the load may require more power than the harvester's output, it is bound to lose power during its computation unless the energy storage buffer is large enough to satisfy its requirements. The storage buffer, which is usually a battery or supercapacitor, accumulates the energy until it can deliver sufficient power to the load. Supercapacitors are well suited for energy harvesting applications, as they provide infinite charge/discharge life cycles, fast recharge rate and high power density compared to batteries. In the above example, the load requires 7.3mW in 12.5s to compute a cryptographic signature, which requires 91mJ of energy. When the load is supplied by a 3V input voltage, it requires a minimum of 0.02F supercapacitor to supply the power required to compute one signature. The number of signatures that can be computed before a power loss occurs depends on the size of the energy buffer which is typically small to reduce the size of the energy harvesting circuit and capacitor charge time. The load experiences a power loss after it exhausts its energy buffer.

Table 5.1: Breakdown of the contents of the checkpoints of our ECDH implementation

Variable	**Size (B)**	**Comments**
System data	763	Device specific data
Generator(G)	96	Memory required to
Shared secret(s_{AB})	96	store an elliptic
Bob's public key (q_B)	96	curve point as a
Alice's public key (q_A)	96	Jacobian coordinate
Alice' private key (d_A)	32	256-bit integer
Order of generator (n)	32	
Total	1211	

5.2.2 Target application

We chose Elliptic Curve Diffie Hellman key exchange (ECDH) as a representative of an application that needs secure checkpoints. ECDH can be used to exchange keys between two entities, Alice and Bob, via an unsecured channel to secure the communication channel in several steps, illustrated in Figure 5.1. First, they agree upon an elliptic curve, E; a base point, G, in E whose order is n. The order n is the smallest integer such that $n.G = 0$. Second, they each chose an integer, N_i, less than n as their corresponding private key, d_i. Third, they compute the product $N_i.G$ as their public key, q_i. Last, each entity generates the shared secret by multiplying its secret with the other entity's public key, which is then used to derive the secret key, KEY. We implement ECDH on MSP430FR5994 using the NIST curve P-256, which provides 128-bits of security, with the help of the RELIC cryptographic library [9]. We consider Alice to be the target microcontroller. ECDH involves long running arithmetic operations on the elliptic curve, such as generating integer and point multiplications. We place `secureCheckpoint()` calls after long running arithmetic operations to ensure the availability of the intermediate results in the event of power loss, as illustrated in Figure 5.1.

Table 5.2: Energy and time overhead of different technology and corresponding duty cycle, D_{load}, when input power is 2 mW

Technology	**Operation**	**Time (ms)**	**Energy (μJ)**	P_{load} **(mW)**	D_{load} **(%)**
Application	ECDH	7800	48300	6.2	33
CTPL	Initialize	0.02	0.03	8.0	25
	Refresh	13.8	12.1		
	Restore	13.5	12.3		
SICP	`INITIALIZE()`	0.06	0.04	9.5	21
	`REFRESH()`	216.2	160.2		
	`RESTORE()`	277.1	202.3		

Checkpoint Location and Contents :

The application specific variables that are required to ensure forward progress of ECDH are listed in Table 5.1. All the variables listed in Table 5.1 are placed in the `.checkpoint` section of tamper-sensitive non-volatile memory using the `__attribute__` keyword, as described in Chapter 3. The generator, G, of the curve is a point on the 256-bit elliptic curve, consisting of three Jacobian coordinates, X, Y, and Z, where each coordinate is 256-bit long. Thus, each point on the elliptic curve, such as the shared secret, s_{AB}, and public keys, q_A and q_B are 96 B. The shared secret, s_{AB}, and Alice's private, d_A, are also checkpointed to maintain secure sessions across power losses. Since d_A and n are 256-bit integers, they only occupy 32 B each. Of the 1211 B of checkpointed data, only G, n, and the public keys, q_A, and q_B, are global public elements, the rest of the variables must be protected from the attacker to maintain the security of the communication channel. Thus, checkpoints of ECDH require SICP to maintain the security properties of its application across stateful power transitions.

5.2.3 Overhead Analysis

The device under test was operated at 1 MHz and was powered by an external power supply. The energy and time measurements reported in this section were measured across a 1 $k\Omega$ shunt resistor using a Tektronix DPO3034 oscilloscope operating at 50 kS/s. Table 5.2 lists the energy and time overhead of ECDH, CTPL, and SICP. It computes the net load power after introducing each facet based on the assumption that the initialize operations occur only once and are ignored and only one checkpoint generation operation is performed during each ON-state.

Energy harvesters : In the continuous execution paradigm, the microcontroller consumes 48.3 mJ of energy in 7.8 s to arrive at the shared secret of an ECDH operation, which requires 6.2 mW of power. Energy harvesters do not always provide the peak power required by the microcontroller, they typically provide a few μW to mW of power [54]. We assume that the microcontroller is powered by a kinetic energy harvester [25], which provides an average power of 2 mW. In the intermittent execution paradigm, the microcontroller still requires 6.2 mW of power to arrive at the shared secret but it operates at only 32% duty cycle as P_{load} is greater than P_{EH}. The microcontroller repeatedly experiences power loss for every 2 mW of power it consumes.

Statefulness : CTPL stores the volatile state information in non-volatile memory as a checkpoint and retrieves it to restore the microcontroller after a power cycle, which introduces overhead. Table 5.1 lists the checkpoint size of our target application as 1211 B, which is calculated by studying the memory section containing $STATE$ [83]. The checkpoint generation and restoration operations combined introduce an overhead of 1.8 mW, listed in Table 5.2. In addition to the power requirements of ECDH operations, the checkpointing

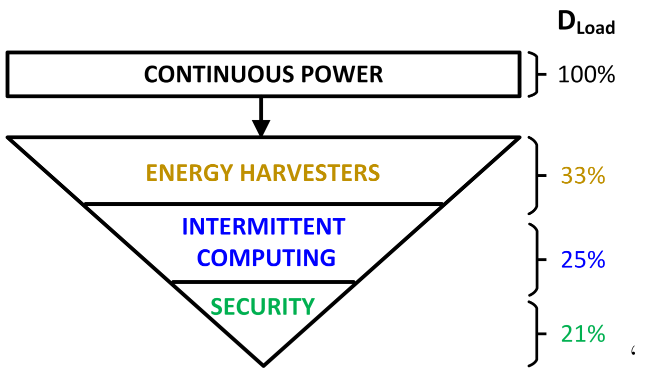


Figure 5.2: Facets of power transition and corresponding effect on duty cycle, D_{load}

overhead increases the net load power, P_{load}, to 8 mW. The microcontroller duty cycle is reduced to 25% to arrive at the shared secret, s_{AB}, assuming that the device is still powered by the same kinetic energy harvester.

Overhead from security : Table 5.2 lists the additional overhead SICP introduces to stateful power transitions. It presents the amount of energy and time required to secure the generation and restoration of 1211 B of checkpointed data. The overhead measurements correspond to `INITIALIZE, REFRESH`, and `RESTORE` operations of the protocol. The microcontroller requires an additional 1.4 mW to secure 1211 B of checkpoint. This increases the net load power, P_{load}, to 9.5 mW. The microcontroller must operate at 21% duty cycle to finish its ECDH operations while ensuring that its security properties and that of ECDH are maintained across power transitions.

Checkpoint size and non-volatile memory In our implementation of secure and stateful power transition, we observe that each facet, including the kinetic energy harvester, CTPL, and SICP, introduces a limiting factor that progressively reduces the duty cycle of the device, illustrated in Figure 5.2. We studied the effects of energy influx, type of non-volatile memory, and checkpoint size on the duty cycle of the load. Figure 5.3 illustrates the

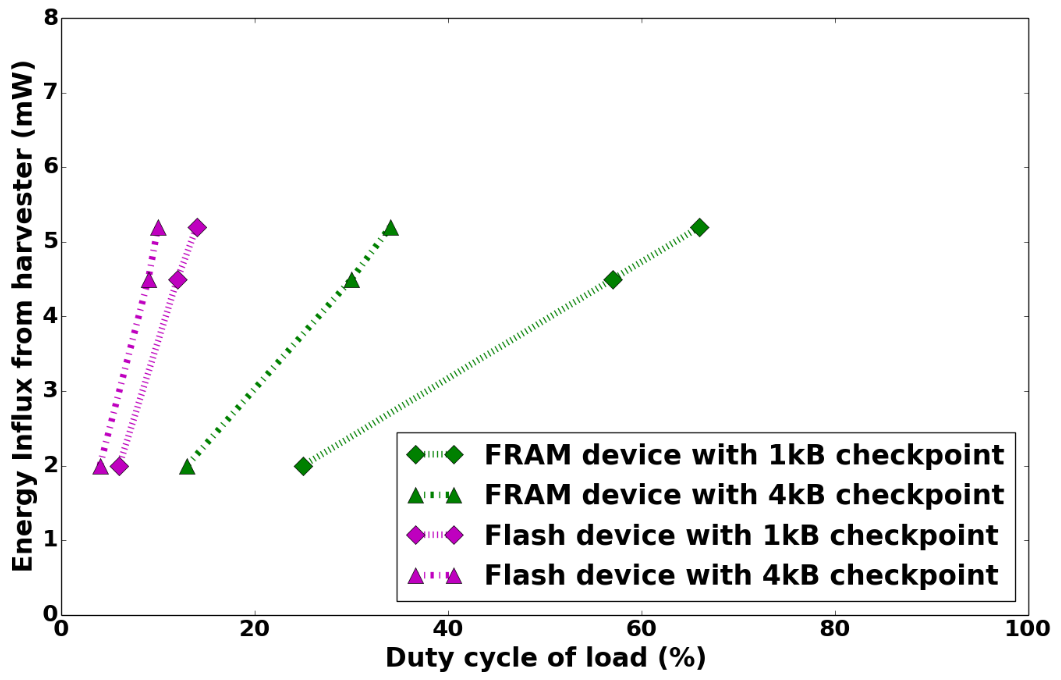


Figure 5.3: Changes to the duty cycle of load, D_{load}, based on the energy influx [25, 56], size of checkpoint, and type of non-volatile memory. The duty cycle measurements were calculated based on the ratings of flash memory and FRAM available in the device datasheet [86, 87]

change in duty cycle based on the energy influx for different checkpoint sizes and non-volatile memories. The data points on each line graph correspond to the duty cycle of the device based on the energy influx from three types of sources, including kinetic, vibration, and thermal harvesters, which provide 2 mW, 4.5mW, and 5.2 mW, respectively [25, 56]. The energy influx varies, between a few μW to a few mW, depending on the choice of the harvester [54]. We chose two types of non-volatile memory commonly available in off-the-shelf devices. First, we studied the flash memory available in TI's MSP432P401R [86]. Second, we studied FRAM available in MSP430FR5994 [87]. Since FRAM consumes low power when compared to flash, the duty cycle of FRAM devices is higher than that of flash devices. While Flash memory is widely available in low-power microcontrollers, it is not suitable for energy harvesting applications because of its additional wait states at higher clock frequency, its low endurance, and its power hungry write operations. We also considered two checkpoint sizes, 1kB and 4kB, to account for applications whose checkpoints may be larger than ECDH's 1.2kB checkpoint. Figure 5.3 illustrates that larger checkpoints reduce the duty cycle of the load, irrespective of the energy influx and type of non-volatile memory in use. The duty cycle reported in Figure 5.3 will reduce further when secure checkpoints are employed. Thus, we must consider the various technologies involved in an energy harvested node, including, but not limited to the energy influx, type of non-volatile memory, application, frequency of checkpoints, and type of device, to achieve the required duty cycle of the target device.

5.3 Effect of Application on Checkpoint Security

The checkpoint properties, such as size, content, and frequency, determine the overhead of securing checkpoint refresh and restore operations. The checkpoint properties are largely determined by the application, microcontroller, and intermittent computing technique used by an IoT device. In this section, we focus on how the contents of the checkpoint are partially dependent on the application and we leverage this dependency to reduce the overhead of securing checkpoints. In this section, we analyze the common checkpoint content and differentiate them with application specific checkpoint content using a curated list of embedded benchmarks. We also briefly describe our experimental setup with the choice of microcontroller and intermittent computing technique used in this work.

5.3.1 System Overview

Target platform: We use Texas Instruments' (TI) MSP430FR5994 LaunchPad Development Kit as a representative of an energy harvested device. MSP430FR5994 is a 16-bit

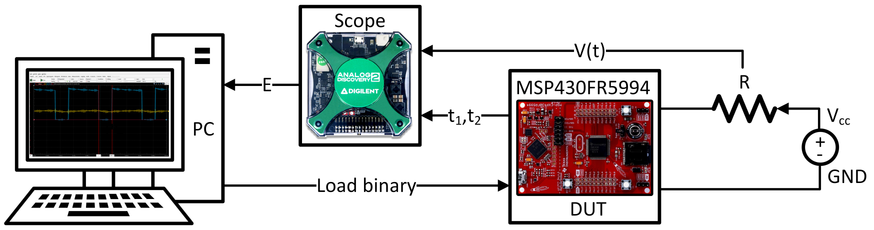
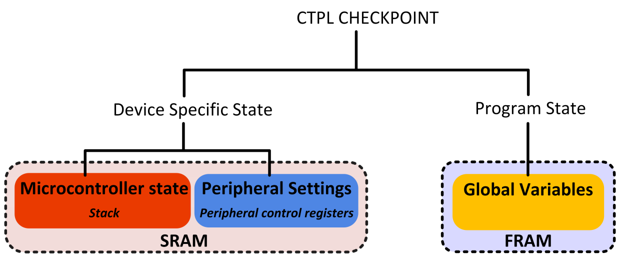


Figure 5.4: Block diagram of the experimental setup. PC is used to load benchmark binaries onto MSP430FR5994 LaunchPad Development Kit which is powered using a 3.5V (V_{CC})DC power supply via a 1kΩ(R) shunt resistor. A Digilent Analog Discovery 2 USB Oscilloscope, which is triggered using the GPIO pins on-chip, is used to capture the voltage across the resistor. Energy measurements computed using Equation 5.2 are sent to PC for logging.

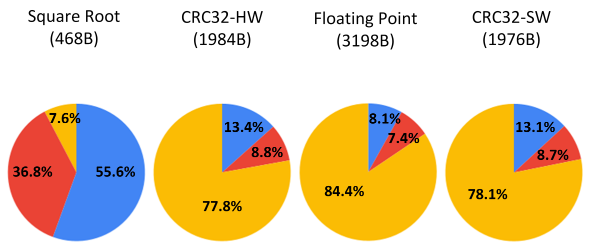
ultra-low power microcontroller that only consumes 120μA/MHz of active current [88]. It is equipped with 256kB of Ferroelectric RAM (FRAM) and 8kB of SRAM. FRAM is a non-volatile storage that retains data even after power loss. When compared to Flash, FRAM has faster write times, lower power consumption, and higher endurance. Apart from its suitability for energy harvesting applications, MSP430FR5994 is equipped with several peripherals, such as CRC32 and AES256, that are useful to accelerate applications. In our evaluation, we use the CRC32 peripheral in a benchmark application to demonstrate the change in security properties based on peripherals used by the application.

$$E = \frac{V_{CC}}{R} \int_{t_1}^{t_2} v(t)\, dt \qquad (5.2)$$

Experimental setup The measurements were collected on MSP430FR5994 LaunchPad Development Kit across a 1 $k\Omega$ shunt resistor using Digilent Analog Discovery 2 USB Oscilloscope. The scope was operated at 1MHz and triggered using on-chip GPIO to identify measurements for target functions. The microcontroller was powered using an external DC power supply at $V_{CC} = 3.5V$, as illustrated in Figure 5.4, and operated at 8MHz using on-chip clock source. The energy consumption of a function's execution is computed using



(a) Checkpoint partition across benchmarks. The device specific space is stored in SRAM or volatile memory at run-time. The program variables are already stored in FRAM. During checkpoint refresh, CTPL only stores the device specific state in FRAM



(b) Variation in individual partitions of CTPL's checkpoint content for square root, CRC32-HW, floating point, and CRC32-SW benchmarks with 468B, 1984B, 3198B, and 1976B of checkpoint data, respectively.

Figure 5.5: Similarities in checkpoint partitions and differences in checkpoint content across benchmarks

equation 5.2, which is a function of the integral of changing voltage across the shunt resistor, R. The difference between t_2 and t_1 is the time taken to execute the target function. The benchmarks were compiled using msp430-gcc 9.2.0 with -O3 optimization.

Intermittent computing: We use TI's Compute Through Power Loss (CTPL) utility for system state restoration after power failure [96]. It is a software utility that triggers checkpoint generation by monitoring V_{cc} using the on-chip analog-to-digital converter (ADC). If CTPL is enabled, the checkpoint, which contains CPU and peripheral states, is automatically saved in FRAM and used for a faster wake-up upon power-up. CTPL takes advantage of the unified memory model of FRAM to directly place constant data and program variables in FRAM.

5.3.2 Benchmark Applications

In our work, the purpose of a benchmark suite is not to evaluate the target platform's performance. Rather, we use the benchmarks to evaluate the different characteristics an application introduces to secure intermittent computing. The characteristics include checkpoint size, checkpoint contents, security level, and energy requirements of both application and secure intermittent computing. The checkpoint size, the number of bytes that must be secured and verified, also determines the overhead of securing checkpoints. The contents of the checkpoint vary based on the application and checkpoint security properties. The net energy consumed just by the application also determines the amount of energy left for secure intermittent computing and the number of checkpoints required to complete the application.

Since our application domain is in energy harvesting devices, we focus on benchmarks for energy measurements, particularly for embedded platforms. We selected ten benchmarks listed in Table 5.3 from MiBench [37] and BEEBS [72] benchmark suites. The set contains a combination of security, mathematical, and signal processing applications. They were originally used to stress the integer, floating point, and memory pipelines; test memory access; and test data caching effects on an embedded platform. Although each benchmark is unique and introduces certain variations to intermittent computing, we first discuss the similarities among them and then focus on the differences. We use the differences to demonstrate the variation in performance cost, checkpoint size, and checkpoint contents across benchmarks, which provides a foundation for configuring checkpoint security based on the needs of the application.

5.3.3 Similarities Among Benchmarks

Checkpoint partition We broadly partition the contents of checkpoints in all benchmarks into device specific state and program variables. Figure 5.5a illustrates each partition and its contents. The device specific state includes peripheral settings and microcontroller state. The peripheral settings contain the control registers, which are stored in SRAM at run-time, that are required for the forward progress of peripherals used by the microcontroller, CTPL, and the benchmark. The microcontroller state, also known as CPU state, contains the stack which is stored in SRAM at run-time, which generates approximately 174B of checkpoint data. The stack includes general purpose registers and the application stack. Since the program variables are already stored in FRAM at run-time, CTPL only stores the device specific state in FRAM during checkpoint generation. For simplicity, we only consider the global variables used by benchmarks as program variables in checkpoints.

Peripheral settings We list the peripherals that require checkpointing based on their usage and their contribution to checkpoint size in bytes for MSP430FR5994. The peripherals that require checkpointing for the regular operation of the microcontroller include memory protection unit (14B), system control state (4B), clock system (12B), FRAM controller (4B), special function reset (4B), GPIO ports (58B) and watch dog timer (2). The peripherals used by CTPL that require checkpointing are analog-to-digital converter (82B), reference voltage generator (2B), and direct memory access (78B). The required peripheral settings for microcontroller and CTPL operation are the same for all benchmarks and sums up to 260B of checkpoint data. The peripherals needed by the benchmark depend on the needs of the application and may contribute a few bytes to the checkpoint.

CTPL overhead CTPL uses a unified memory model where a majority of data required for forward progress is always stored in non-volatile memory. At run-time, only the device specific state, which is volatile, requires to be checkpointed, i.e., written into non-volatile memory. As the name suggests, the device specific state mostly contains checkpoint data required for restoring the microcontroller, peripherals, and a few volatile application variables. Table 5.3 lists the size of device specific state for all benchmarks and the energy and time required to create and restore a checkpoint of device specific state under CTPL overhead. The

Table 5.3: The variation in benchmark cost, checkpoint size, and contents; and the similarities in CTPL overhead and device specific state across different benchmark applications. The benchmark cost includes the energy and time taken to execute one iteration of a benchmark function. The CTPL overhead presents the energy and time taken to refresh and restore an unsecure checkpoint using CTPL. Checkpoint size is partitioned into program variables and device specific state, which typically consists of microcontroller state and peripheral settings.

	Benchmark cost		**CTPL overhead**		**Checkpoint size**		
Benchmark	**Energy (μJ)**	**Time (ms)**	**Energy (μJ)**	**Time (ms)**	**PV (B)**	**DSS (B)**	**Total(B)**
Binary Search	10.66	3.98	2.6	0.9	200	436	636
Dijkstra	507.60	178.00	2.2	0.9	253	434	687
Exponent	0.01	0.02	2.3	0.9	60	492	552
Hash Table	0.18	0.06	1.8	0.6	2416	432	2848
Floating Point	6986.00	2605.00	1.3	0.5	2700	498	3198
Square Root	1.51	0.59	2.5	0.9	36	432	468
Binary Tree	6.81	2.53	2.4	0.9	380	432	812
SHA-2	388.60	147.52	2.6	0.9	100	532	632
CRC32-SW	54.91	20.05	2.2	0.7	1544	432	1976
CRC32-HW	1.38	0.49	2.5	0.9	1810	174	1984

PV: Program Variables, DSS: Device Specific State

measurements were computed by placing checkpoint calls at the boundaries of benchmark functions to capture necessary global variables in checkpoints. The checkpoint calls may also be placed within benchmark functions to capture local variables in the checkpoint which may change the frequency, and overhead of generation and restoration of checkpoints. We observed similar overhead for checkpointing across all the benchmarks which are attributed to the similarity in device specific state sizes. In our experiments, on average, the checkpoint generation and restoration for benchmarks consumed 2.2mJ of energy and introduced 0.8s latency.

5.3.4 Differences Among Benchmarks

Device specific state Even though peripheral settings and microcontroller state are *device specific*, they also contain application specific content such as the application stack and peripherals required by benchmarks. Thus, there may be variations in device specific state depending on the benchmarks. For example, an application may use a larger stack or use other peripherals such as the CRC32 peripheral used in the CRC32-HW benchmark which adds an additional 6B to the total checkpoint size when compared to CRC32-SW benchmark. Figure 5.5b illustrates the variation in checkpoint content for a few selected benchmarks using the data provided in Table 5.3. The peripheral settings contribute to the majority of checkpoint content in the square root benchmark, whereas, the program state makes up for over 75% of the checkpoint content for CRC32-HW, floating point, and CRC32-SW benchmarks. The small variation in device specific state measurements in Table 5.3 were only caused by microcontroller state, i.e. application stack, in all benchmarks except CRC32-HW, which is described below.

Program variables In CTPL, the checkpoints only contain device specific state. Although the program variables are not checkpointed by CTPL upon detecting power loss, they are a part of the checkpointed state and need security guarantees. Table 5.3 lists the size of the overall checkpoint and its broad partitions which helps visualize the dependency between checkpoint size and benchmarks. While the device specific checkpoint state is mostly similar across benchmarks, the program variables content vastly varies among benchmarks. For example, the square root benchmark only checkpoints 36B of program variables, whereas the floating point benchmark checkpoints 2700B of program variables. For CRC32-HW benchmark, we consider the peripheral settings to be part of the program variable as the CRC32 peripheral processes inputs from the benchmark. This reduces the device specific

	SL4		**SL3**		**SL2**		**SL1**	
Benchmark	**Energy (μJ)**	**Time (ms)**	**Energy (μJ)**	**Time (ms)**	**Energy (μJ)**	**Time (ms)**	**Energy (μJ)**	**Time (ms)**
Binary Search	2.6	0.9	98.9	35.9	163.4	61.3	243.2	91.5
Dijkstra	2.2	0.9	64.8	23.2	135.6	68.3	263.5	98.3
Exponent	2.3	0.9	85.6	29.9	138.6	49.1	183.6	67.9
Hash Table	0.8	0.6	181.2	65.5	836.6	313.6	911.8	346.2
Floating Point	1.3	0.5	94.0	38.7	895.9	336.5	1014.0	379.9
Square Root	2.5	0.9	132.1	48.9	140.6	51.8	526.7	200.5
Binary Tree	2.4	0.9	117.9	42.5	241.5	89.2	319.5	120.9
SHA-2	2.6	0.9	54.8	19.9	164.6	58.6	268.6	101.2
CRC-SW	2.2	0.7	202.3	72.5	642.7	240.0	727.8	274.1
CRC32-HW	2.5	0.9	153.8	72.4	721.8	270.0	728.6	273.5

Table 5.4: Performance overhead of generating and restoring checkpoints securely using multi-level security based optimized SICP implementing in various benchmarks

state to just the microcontroller state. The partition in checkpoint size also highlights the need for individualized security properties required by different sections of checkpoints.

Benchmark cost Table 5.3 lists the energy and time required to complete one iteration of each benchmark under benchmark cost. The energy consumed by each benchmark function depends on certain application specific features, such as the type of input (integer/float), the size of the input, the number of iterations performed by each benchmark, and the benchmark itself. We added CRC32-HW benchmark to demonstrate the variation in benchmark overhead when on-chip peripherals are in use. CRC32-HW uses the CRC32 peripheral on MSP430FR5994 to improve the performance of software-only CRC verification (CRC32-SW). As expected, the hardware accelerated benchmark outperforms the software-only benchmark for CRC32 with 40x improvement. The variation in the performance overhead of each benchmark demonstrates the change in energy requirement for each application which is elaborated in Section 5.3.5.

5.3.5 Results

The measurements reported in this section were measured on the same experimental setup described in Section 5.3.1. We use energy and time as metrics to evaluate the overhead of different levels of security. The energy overhead helps understand the overall energy required in securing an energy harvested application where the input energy is limited. The time overhead helps understand the latency that secure checkpoints may introduce to the application. We measure the overhead of generating and restoring one checkpoint at all four security levels. Table 5.4 lists the energy and time overhead of securely generating and restoring checkpoints using SL4, SL3, SL2, and SL1. In all implementations but CRC32-HW, we place the peripheral registers as a part of device specific state in public memory in SL2. For CRC32-HW, we place the peripheral registers as a part of program variables. The program variables of all the benchmarks are placed in the private section for SL1 and SL2, whereas in SL3 there is no private section as there is no confidentiality guarantee. The overhead of checkpointing at SL4 is also listed in Table 5.3, as SL4 is equivalent to unsecure intermittent computing. The overhead listed for each level includes secure checkpoint generation and restoration.

Improvements With Multi-level Security

Figure 5.6 illustrates the n-fold increase in energy required to securely generate and restore one checkpoint at different security levels for our benchmarks. The increase was calculated with the unsecured checkpointing overhead listed under SL4 in Table 5.3 as baseline energy consumption. A similar trend in the n-fold increase in time for securing checkpoints was observed across the benchmarks. SL3 has the least increase in energy consumption as there is no encryption/decryption involved in securing checkpoints. Mostly, benchmarks with larger

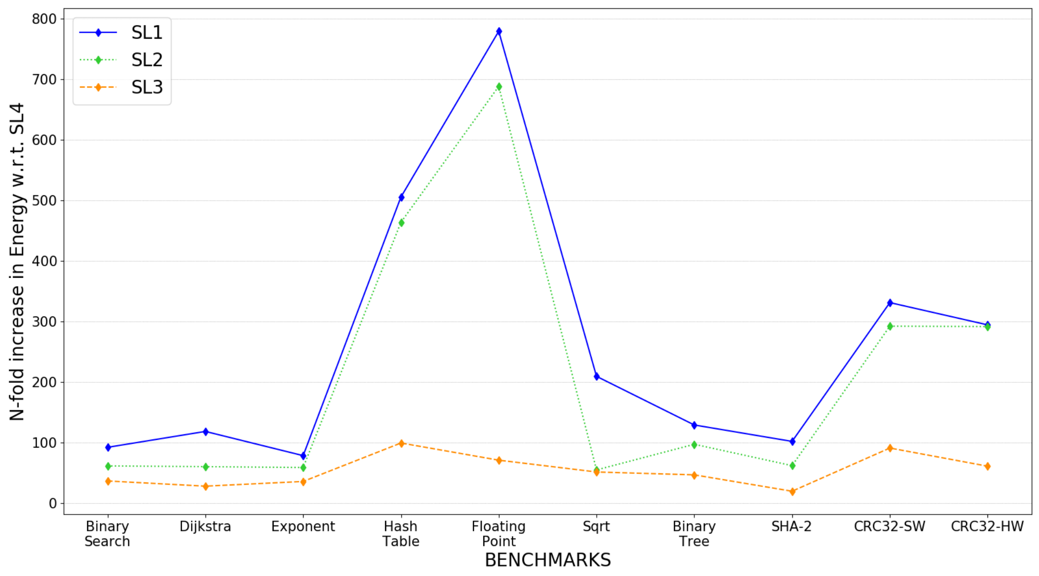


Figure 5.6: N-fold increase in energy required to secure checkpoints of various benchmarks when operated at different security levels (SL1-3) with respect to (w.r.t.) the energy required for unsecure checkpoints (SL4)

checkpoints, such as floating point, CRC32, and hash table, consume significantly more energy across all levels of security when compared to benchmarks with smaller checkpoints(smaller than 1000B). Also, in large checkpoint benchmarks, there is a significant increase in energy consumption at SL2 and SL1 as the size of secure memory (program variables) is correspondingly large. This illustrates that the overhead of checkpoint security is vastly dependent on the application and the programmer needs to carefully select security levels based on the contents of the checkpoint to avoid unnecessary overhead incurred from encryption/decryption.

Number Of Checkpoints

The number of checkpoints required for each benchmark varies depending on the size of the energy buffer, input power, security level, and benchmark itself. We hypothesize the charge-

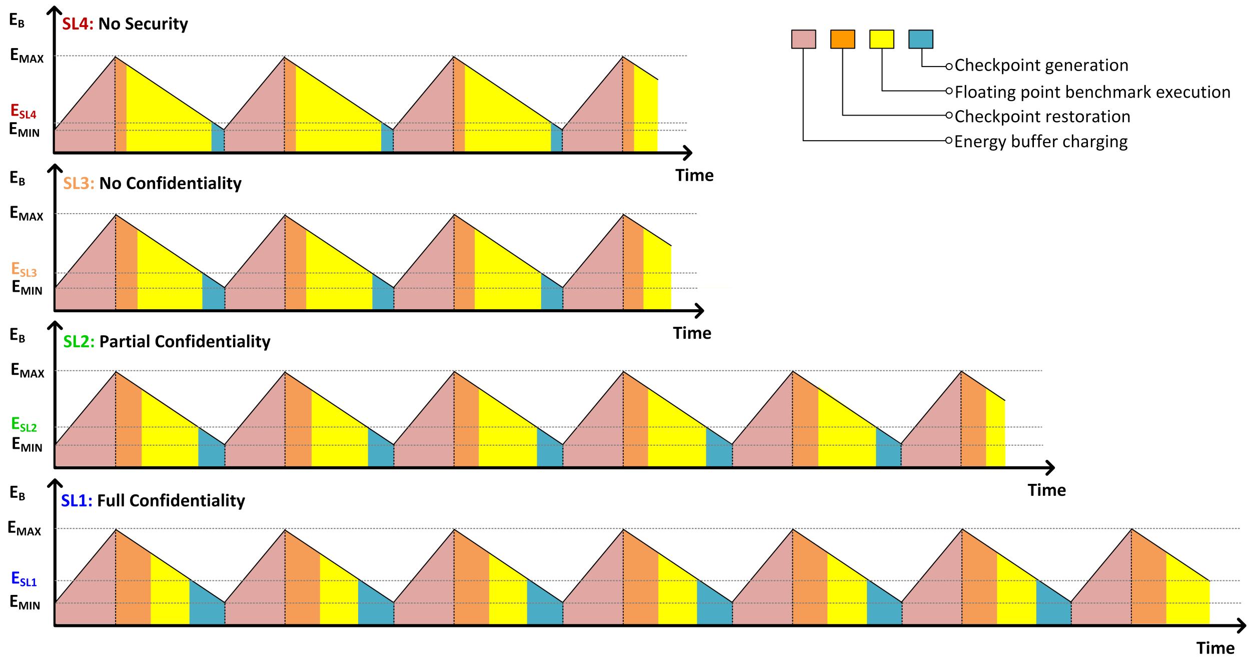


Figure 5.7: A hypothetical power cycle graph of an ideal 470 μF supercapacitor buffer used to complete one iteration of the floating point arithmetic benchmark. E_{max}=2.1mJ, E_{SL4}=1.3μJ, E_{SL3}=0.09mJ, E_{SL2}=0.9mJ, and E_{SL1}=1.01mJ. With the increase in security level from SL4 to SL1, the amount of energy available for the forward progress of the benchmark during each power cycle is reduced to accommodate securely refreshing and restoring checkpoints, which in turn increases the number of checkpoints required to complete one iteration of the benchmark. This graph does not consider the idle time spent by the supercapacitor in waiting for input from the energy harvester.

discharge cycles of an ideal 470 μF supercapacitor in Figure 5.7 to demonstrate the change in input energy requirement with change in security levels for intermittent computing. If we consider a 470 μF supercapacitor as an energy buffer [92], it can provide 2.1 mJ of energy in one power cycle when the input voltage to the supercapacitor is 3V. Let us consider the floating point and SHA-2 benchmark. For SHA-2, 470 μF supercapacitor provides sufficient input energy to complete more than one iteration of the benchmark without power loss, as SHA-2 only consumes 0.38 mJ of energy. Whereas, the floating point benchmark consumes 6.9 mJ of energy for one iteration and the microcontroller may require at least four checkpoints to complete the benchmark if the supercapacitor is not continuously charged. Apart from the checkpointing overhead and benchmark cost, the different levels of checkpoint security incur additional overhead based on the level, i.e, from SL3 to SL1 each secure checkpoint generation and restoration consumes an additional 0.09, 0.89, and 1.1 mJ of energy. And, there is a corresponding increase in the number of checkpoints or power cycles across different levels. Figure 5.7 illustrates that customizing checkpoint security policy based on the contents of the checkpoint may help reduce the number of checkpoints required to securely finish a benchmark, which ultimately improves the performance of the benchmark. It also illustrates that an energy harvesting system must be designed with careful consideration to the choice of energy buffer and energy harvester to ensure forward progress of the application with minimum latency. The energy harvester determines the amount of energy available to charge the energy buffer. And, the energy buffer limits the amount of energy available to the microcontroller in the event of power loss from the energy harvester.

Estimation of energy and time requirement

Based on our experiments, we estimate the amount of time and energy required per byte to add security to checkpoints. The following average values were computed for our experimental setup. A similar estimate may be derived for a different operating frequency, input voltage, and device under test. To provide authenticated encryption that satisfies SL1 security properties using optimized SICP, the microcontroller requires approximately 0.18μJ of energy and 0.06ms of time per byte of the checkpoint. Whereas, to only provide SL3 security properties, each byte of checkpoint approximately requires 0.07μJ of energy and 0.03ms of time. One can estimate the energy and time required for providing SL2 security properties by combining the overhead of SL1 and SL3 level security properties for the *Pri* and *Pub* sections of the checkpoint, respectively.

5.4 Conclusions

As energy harvested IoT devices become increasingly common, we need to systematically evaluate the requirement and overhead of secure intermittent computing based on the needs of the application. The introduction of intermittent computing and its security affects the duty cycle of the target application. Our evaluation of ECDH demonstrates the need for careful design choices, including but not limited to non-volatile memory, low power device, cryptographic hardware, and secure memory, to improve the performance of the application. We compiled a benchmark of ten embedded applications to demonstrate the need for customized checkpoint security solutions that is not available in the state-of-the-art secure intermittent computing solutions. Based on our results, we conclude that application plays a vital role in deciding, both, the security properties of the checkpoints and the overhead of secure intermittent computing.

Chapter 6

Conclusions and Future Work

This dissertation presents a top-down approach to designing secure intermittent systems. Foremost, in Chapter 2 we identified the threats involved in intermittent computing and its checkpoints. We identified vulnerabilities that arise from power loss, which was originally not considered a threat to embedded systems. We demonstrated risks and attacks that exploit the risks to show the need for employing secure power transitions in embedded systems. In our research, we identify three key checkpoint vulnerabilities - snooping, spoofing, and replay, which were used to exploit software AES to extract the secret key from memory. On the same line, checkpoints of intermittent systems may be used to attack other energy harvested systems.

In Chapter 3, we identified checkpoint security requirements to overcome security vulnerabilities. At a minimum, the checkpoint needs freshness and authenticity guarantees to detect replay and integrity guarantees to detect checkpoint modifications. Apart from information security guarantees, the availability of checkpoints is an important security requirement that ensures the intermittent system is always left with a valid secure checkpoint. The availability of a checkpoint prevents an attacker from arbitrarily restarting the intermittent application. Originally, the confidentiality of checkpoints was a minimum security requirement. Later, the confidentiality of checkpoints was included as an optional security requirement. Based on these requirements, we designed the Secure Intermittent Computing Protocol as a fail safe way to incorporate security into a checkpoint.

Based on preliminary evaluation, we identify optimizations for secure intermittent systems in Chapter 4. The intermittent application plays an important role in defining security requirements for checkpoints. As the contents of checkpoints change based on the application, we believe the checkpoint security requirements must also change based on the application. While the minimum security requirement remains the same across the entire checkpoint, we propose to add-on checkpoint encryption based on the needs of confidentiality for different parts of checkpoints. We propose to configure checkpoint security into multiple levels, SL1-SL4, based on the needs of the application. We also propose protocol specific optimization to improve the performance of the first iteration of SICP.

Finally, we identify various factors that must be considered in designing a secure intermittent system in Chapter 5. In particular, we study the interdependency between application and checkpoint security. Apart from the application, we study other factors, including the type of energy harvester which controls the input power, the type of non-volatile memory which determines the overhead of reading from and writing to non-volatile memory, and the type of intermittent computing technique to determine the contents and frequency of checkpoints. We demonstrate their importance using the application duty cycle as a metric of performance. We curated a list of benchmarks that were used to evaluate the advantages of multi-level security in secure intermittent systems. Based on the results of this study, we conclude that application plays a major role in the design of intermittent systems.

In this dissertation, we provided a foundation for designing secure intermittent systems starting from threat discovery, followed by secure design, system optimization, and ending in the evaluation of factors that affect the performance of intermittent systems. While the conclusion of each chapter of this dissertation provides incremental future work, here we consolidate a few of the future work and provide tentative directions towards securing intermittent systems beyond this dissertation:

- **Optimization**: There are several avenues for optimization of security in intermittent systems. First, we may consider automation of classifying secure sections of checkpoints. In this dissertation, we provided a high-level division of checkpoints using programmer intervention to identify checkpoint security properties. In the future, an automatic tool may be built using, for example using LLVM, to identify all uses of a secure variable to ensure only necessary variables are encrypted to prevent unnecessary encryption overhead. Second, we may consider hardware acceleration for both the latest lightweight cryptographic algorithms such as the NIST LWC candidates and for DMA assisted checkpoint generation and restoration process. Such hardware acceleration will reduce the overhead of intermittent computing and its security which allocates more of the input power towards the forward progress of the application.

- **Secure Energy Estimation**: The intermittent systems understand power loss based on external input from the energy harvester. So far, the estimation of energy available in the energy buffer is assumed to be trusted. But since this input is external to the system, it may be tampered by an attacker with access to the intermittent system. To comprehensively secure energy harvester devices, we must identify threats present in communicating the energy estimation from energy harvester and arrive at solutions to ensure intermittent systems are provided with secure energy estimation.

We proposed a few potential research directions based on our research questions and our results. There may be other research directions that could provide additional features to secure intermittent systems, such as designing a hardware interface with secure energy estimation to trigger the generation of secure checkpoints upon detecting power loss from a trusted source. This dissertation may be considered as a stepping stone to designing and utilizing secure intermittent systems.

Bibliography

[1] Atmel Flash Microcontrollers. https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-4099-Flash-Microcontrollers-Product-Portfolio_Brochure.pdf.

[2] STM32F10xxx Flash memory microcontrollers. https://www.st.com/resource/en/programming_manual/cd00283419-stm32f10xxx-flash-memory-microcontrollers-stmicroelec pdf.

[3] Zatara High-Performance, Secure, 32-Bit ARM Microcontroller. Technical report, Maxim, March 2009. URL https://datasheets.maximintegrated.com/en/ds/ZA9L1.pdf.

[4] *MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family User's Guide*. Number SLAU678M. 2012. URL http://www.ti.com/lit/ug/slau367o/slau367o.pdf.

[5] *SR6 P7 line of Stellar integration MCUs*. Number DB4504. 2021.

[6] H. Afzali-Kusha, A. Shafaei, and M. Pedram. 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)*, pages 280–285, March 2018. doi: 10.1109/ISQED.2018.8357301.

[7] M. Agoyan, J. M. Dutertre, A. P. Mirbaha, D. Naccache, A. L. Ribotta, and A. Tria. Single-bit DFA using multiple-byte laser fault injection. In *2010 IEEE International Conference on Technologies for Homeland Security (HST)*, pages 113–119, Nov 2010. doi: 10.1109/THS.2010.5655079.

[8] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Efficient intermittent computing with differential checkpointing. In Jian-Jia Chen and Aviral Shrivastava, editors, *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2019, Phoenix, AZ, USA, June 23-23, 2019*, pages 70–81. ACM, 2019. doi: 10.1145/3316482.3326357. URL https://doi.org/10.1145/3316482.3326357.

[9] Diego F Aranha and Conrado Porto Lopes Gouvêa. RELIC is an Efficient LIbrary for Cryptography, 2010.

[10] Alberto Rodriguez Arreola, Domenico Balsamo, Geoff V. Merrett, and Alex S. Weddell. RESTOP: retaining external peripheral state in intermittently-powered sensor systems. *Sensors*, 18(1):172, 2018. doi: 10.3390/s18010172. URL https://doi.org/10.3390/s18010172.

[11] Hafiz Areeb Asad, Erik Henricus Wouters, Naveed Anwar Bhatti, Luca Mottola, and Thiemo Voigt. On securing persistent state in intermittent computing. In *Proceedings of the 8th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems, ENSsys@SenSys 2020, Virtual Event, Japan, November 16, 2020*, pages 8–14. ACM, 2020. doi: 10.1145/3417308.3430267. URL https://doi.org/10.1145/3417308.3430267.

[12] Subhadeep Banik, Avik Chakraborti, Tetsu Iwata, Kazuhiko Minematsu, Mridul Nandi, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT-COFB v1.0. Submission to Round 2 of the NIST Lightweight Cryptography project, 2019. URL https://csrc.nist.gov/CSRC/media/

Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/gift-cofb-spec-round2.pdf.

[13] A. Barenghi, G. M. Bertoni, L. Breveglieri, M. Pellicioli, and G. Pelosi. Fault attack on AES with single-bit induced faults. In *2010 Sixth International Conference on Information Assurance and Security*, pages 167–172, Aug 2010. doi: 10.1109/ISIAS.2010.5604061.

[14] S. C. Bartling, S. Khanna, M. P. Clinton, S. R. Summerfelt, J. A. Rodriguez, and H. P. McAdams. An 8MHz 75 microa/MHz zero-leakage non-volatile logic-based cortex-m0 mcu soc exhibiting 100lt;400ns wakeup and sleep transitions. In *ISCC2013*, pages 432–433, Feb 2013. doi: 10.1109/ISSCC.2013.6487802.

[15] Mihir Bellare, Phillip Rogaway, and David A. Wagner. The EAX mode of operation. In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, volume 3017 of *Lecture Notes in Computer Science*, pages 389–407. Springer, 2004. doi: 10.1007/978-3-540-25937-4_25. URL https://doi.org/10.1007/978-3-540-25937-4_25.

[16] N. Beringuier-Boher, K. Gomina, D. Hely, J. B. Rigaud, V. Beroulle, A. Tria, J. Damiens, P. Gendrier, and P. Candelier. Voltage glitch attacks on mixed-signal systems. In *2014 17th Euromicro Conference on Digital System Design*, pages 379–386, Aug 2014. doi: 10.1109/DSD.2014.14.

[17] Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. Peripheral state persistence for transiently-powered systems. In *Global Internet of Things Summit, GIoTS 2017, Geneva, Switzerland, June 6-9, 2017*, pages 1–6. IEEE, 2017. doi: 10.1109/GIOTS.2017.8016243. URL https://doi.org/10.1109/GIOTS.2017.8016243.

[18] Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. Sytare: A lightweight kernel for NVRAM-based transiently-powered systems. *IEEE Trans. Computers*, 68(9):1390–1403, 2019. doi: 10.1109/TC.2018.2889080. URL https://doi.org/10.1109/TC.2018.2889080.

[19] Gautier Berthou, Pierre-Évariste Dagand, Delphine Demange, Rémi Oudin, and Tanguy Risset. Intermittent computing with peripherals, formally verified. In Jingling Xue and Changhee Jung, editors, *Proceedings of the 21st ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2020, London, UK, June 16, 2020*, pages 85–96. ACM, 2020. doi: 10.1145/3372799.3394365. URL https://doi.org/10.1145/3372799.3394365.

[20] Joseph Birr-Pixton. Cifra: Cryptographic Primitive Collection. https://github.com/ctz/cifra, 2017.

[21] S. Biswas and S. Neogy. Secure checkpointing using public key cryptography in mobile computing. In *2011 Fifth IEEE International Conference on Advanced Telecommunication Systems and Networks (ANTS)*, pages 1–3, Dec 2011. doi: 10.1109/ANTS.2011.6163669.

[22] G. Bronevetsky, D. Marques, K. Pingali, S. McKee, and R. Rugina. Compiler-enhanced incremental checkpointing for openmp applications. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, May 2009. doi: 10.1109/IPDPS.2009.5160999.

[23] M. Z. Chaari, M. Lahiani, and H. Ghariani. Energy harvesting from electromagnetic radiation emissions by compact flouresent lamp. In *2017 Ninth International Conference on Advanced Computational Intelligence (ICACI)*, pages 272–275, Feb 2017. doi: 10.1109/ICACI.2017.7974520.

[24] Siddhartha Chhabra and Yan Solihin. i-NVMM: A secure non-volatile main memory system with incremental encryption. pages 177–188, 2011. doi: 10.1145/2000064.2000086.

[25] Y. Da and A. Khaligh. Hybrid offshore wind and tidal turbine energy harvesting system with independently controlled rectifiers. In *2009 35th Annual Conference of IEEE Industrial Electronics*, pages 4577–4582, Nov 2009. doi: 10.1109/IECON.2009.5414866.

[26] Joan Daemen and Vincent Rijmen. Rijndael for AES. In *AES Candidate Conference*, pages 343–348, 2000.

[27] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. HAFIX: hardware-assisted flow integrity extension. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 74:1–74:6, 2015. doi: 10.1145/2744769.2744847. URL https://doi.org/10.1145/2744769.2744847.

[28] Daniel Dinu, Archanaa S. Krishnan, and Patrick Schaumont. SIA: secure intermittent architecture for off-the-shelf resource-constrained microcontrollers. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2019, McLean, VA, USA, May 5-10, 2019*, pages 208–217, 2019. doi: 10.1109/HST.2019.8740834. URL https://doi.org/10.1109/HST.2019.8740834.

[29] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. Submission to Round 1 of the NIST Lightweight Cryptography project, 2019. URL https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/ascon-spec.pdf.

[30] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. Differential fault analysis on A.E.S. 2003:293–306, 01 2003.

[31] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

[32] Zahra Ghodsi, Siddharth Garg, and Ramesh Karri. Optimal checkpointing for secure intermittently-powered iot devices. In Sri Parameswaran, editor, *2017 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2017, Irvine, CA, USA, November 13-16, 2017*, pages 376–383. IEEE, 2017. doi: 10.1109/ICCAD.2017.8203802. URL https://doi.org/10.1109/ICCAD.2017.8203802.

[33] S. Ghosh and A. Chakrabarty. Green energy harvesting from ambient RF radiation. In *2016 International Conference on Microelectronics, Computing and Communications (MicroCom)*, pages 1–4, Jan 2016. doi: 10.1109/MicroCom.2016.7522490.

[34] Christophe Giraud. DFA on AES. In *Advanced Encryption Standard – AES*, pages 27–41, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31840-8.

[35] Mikhail I. Gofman, Ruiqi Luo, Ping Yang, and Kartik Gopalan. SPARC: A Security and Privacy Aware Virtual Machinecheckpointing Mechanism. In *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society*, WPES '11, pages 115–124, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1002-4. doi: 10.1145/2046556.2046571. URL http://doi.acm.org/10.1145/2046556.2046571.

[36] Le Guan, Jingqiang Lin, Ziqiang Ma, Bo Luo, Luning Xia, and Jiwu Jing. Copker: A Cryptographic Engine Against Cold-Boot Attacks. *IEEE Trans. Dependable Sec. Comput.*, 15(5):742–754, 2018. doi: 10.1109/TDSC.2016.2631548. URL https://doi.org/10.1109/TDSC.2016.2631548.

[37] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, 2001. doi: 10.1109/WWC.2001.990739.

[38] M. Habibzadeh, M. Hassanalieragh, A. Ishikawa, T. Soyata, and G. Sharma. Hybrid solar-wind energy harvesting for embedded applications: Supercapacitor-based system architectures and design tradeoffs. *IEEE Circuits and Systems Magazine*, 17(4):29–63, Fourthquarter 2017. ISSN 1531-636X. doi: 10.1109/MCAS.2017.2757081.

[39] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 45–60, 2008.

[40] Clemens Helfmeier, Dmitry Nedospasov, Christopher Tarnovsky, Jan Starbug Krissler, Christian Boit, and Jean-Pierre Seifert. Breaking and entering through the silicon. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 733–744, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516717. URL http://doi.acm.org/10.1145/2508859.2516717.

[41] Matthew Hicks. Clank: Architectural support for intermittent computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 228–240, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080238. URL http://doi.acm.org/10.1145/3079856.3080238.

[42] Daniel E. Holcomb, Wayne P. Burleson, and Kevin Fu. Power-up SRAM state as an

identifying fingerprint and source of true random numbers. *IEEE Trans. Computers*, 58(9):1198–1210, 2009. doi: 10.1109/TC.2008.212. URL https://doi.org/10.1109/TC.2008.212.

[43] IEEE. IEEE 802.15.4-2003 - IEEE Standard for Telecommunications and Information Exchange Between Systems - LAN/MAN Specific Requirements - Part 15: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (WPAN). URL https://standards.ieee.org/standard/802_15_4-2003.html.

[44] Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee, and Vijay Raghunathan. Quick-recall: A HW/SW approach for computing across power cycles in transiently powered computers. *JETC*, 12(1):8:1–8:19, 2015. doi: 10.1145/2700249. URL https://doi.org/10.1145/2700249.

[45] P. Jokic and M. Magno. Powering smart wearable systems with flexible solar energy harvesting. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, May 2017. doi: 10.1109/ISCAS.2017.8050615.

[46] Evangelos Kallitsis, Anna Korre, Geoff Kelsall, Magdalena Kupfersberger, and Zhenggang Nie. Environmental life cycle assessment of the production in china of lithium-ion batteries with nickel-cobalt-manganese cathodes utilising novel electrode chemistries. *Journal of Cleaner Production*, 254:120067, 2020. ISSN 0959-6526. doi: https://doi.org/10.1016/j.jclepro.2020.120067. URL https://www.sciencedirect.com/science/article/pii/S0959652620301141.

[47] S. Kannan, N. Karimi, O. Sinanoglu, and R. Karri. Security vulnerabilities of emerging nonvolatile main memories and countermeasures. *IEEE Transactions on Computer-*

Aided Design of Integrated Circuits and Systems, 34(1):2–15, Jan 2015. ISSN 0278-0070. doi: 10.1109/TCAD.2014.2369741.

[48] Muhammad Nauman Khan, Asha Rao, and Seyit Camtepe. Lightweight cryptographic protocols for iot-constrained devices: A survey. *IEEE Internet Things J.*, 8(6):4132–4156, 2021. doi: 10.1109/JIOT.2020.3026493. URL https://doi.org/10.1109/JIOT.2020.3026493.

[49] Sudhanshu Khanna, Steven Bartling, Michael Clinton, Scott R. Summerfelt, John A. Rodriguez, and Hugh P. McAdams. An fram-based nonvolatile logic MCU soc exhibiting 100% digital state retention at $VDD = 0$ V achieving zero leakage with $<$ 400-ns wakeup time for ULP applications. *J. Solid-State Circuits*, 49(1):95–106, 2014. doi: 10.1109/DTIS.2018.8368561. URL https://doi.org/10.1109/JSSC.2013.2284367.

[50] Paul Kocher, Joshua Jaffe, and Benjamin Jun. *Differential Power Analysis*, pages 388–397. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-540-48405-9. doi: 10.1007/3-540-48405-1_25. URL https://doi.org/10.1007/3-540-48405-1_25.

[51] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah D. Hester, and Przemyslaw Pawelczak. Time-sensitive intermittent computing meets legacy software. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 85–99. ACM, 2020. doi: 10.1145/3373376.3378476. URL https://doi.org/10.1145/3373376.3378476.

[52] Archanaa S. Krishnan and Patrick Schaumont. Exploiting security vulnerabilities in intermittent computing. In Anupam Chattopadhyay, Chester Rebeiro, and Yuval Yarom, editors, *Security, Privacy, and Applied Cryptography Engineering - 8th*

International Conference, SPACE 2018, Kanpur, India, December 15-19, 2018, Proceedings, volume 11348 of *Lecture Notes in Computer Science*, pages 104–124. Springer, 2018. doi: 10.1007/978-3-030-05072-6_7. URL https://doi.org/10.1007/978-3-030-05072-6_7.

[53] Archanaa S. Krishnan, Charles Suslowicz, Daniel Dinu, and Patrick Schaumont. Secure intermittent computing protocol: Protecting state across power loss. In Jürgen Teich and Franco Fummi, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, pages 734–739. IEEE, 2019. doi: 10.23919/DATE.2019.8714997. URL https://doi.org/10.23919/DATE.2019.8714997.

[54] M. Ku, W. Li, Y. Chen, and K. J. Ray Liu. Advances in Energy Harvesting Communications: Past, Present, and Future Challenges. *IEEE Communications Surveys Tutorials*, 18(2):1384–1412, Secondquarter 2016. ISSN 1553-877X. doi: 10.1109/COMST.2015.2497324.

[55] Hans-E. Lange, Dennis Hohlfeld, Rainer Bader, and Daniel Kluess. A piezoelectric energy harvesting concept for an energy-autonomous instrumented total hip replacement. *Smart Materials and Structures*, 29, 2020. URL SmartMaterialsandStructures.

[56] Jiayu Li, Ji HoonHyun, and Dong SamHa. A multi-source energy harvesting system to power microcontrollers for cryptography. In *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society, Washington, DC, USA, October 21-23, 2018*, pages 901–906, 2018. doi: 10.1109/IECON.2018.8591833. URL https://doi.org/10.1109/IECON.2018.8591833.

[57] Xia Li, Zhiyuan Li, Cheng Bi, Benxue Liu, and Yufeng Su. Study on wind energy harvesting effect of a vehicle-mounted piezo-electromagnetic hybrid energy harvester.

IEEE Access, 8:167631–167646, 2020. doi: 10.1109/ACCESS.2020.3023649. URL https://doi.org/10.1109/ACCESS.2020.3023649.

[58] Yichen Li, Tianxing Li, Ruchir A. Patel, Xing-Dong Yang, and Xia Zhou. Self-Powered Gesture Recognition with Ambient Light. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18. ACM, 2018. doi: 10.1145/3242587.3242635.

[59] Yunjia Li, Jiaxing Li, Aijun Yang, Yong Zhang, Baoxiang Jiang, and Dayong Qiao. Electromagnetic vibrational energy harvester with microfabricated springs and flexible coils. *IEEE Trans. Ind. Electron.*, 68(3):2684–2693, 2021. doi: 10.1109/TIE.2020.2973911. URL https://doi.org/10.1109/TIE.2020.2973911.

[60] X. Lu, P. Wang, D. Niyato, D. I. Kim, and Z. Han. Wireless Networks With RF Energy Harvesting: A Contemporary Survey. *IEEE Communications Surveys Tutorials*, 17(2): 757–789, Secondquarter 2015. ISSN 1553-877X. doi: 10.1109/COMST.2014.2368999.

[61] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 575–585, 2015. doi: 10.1145/2737924.2737978. URL https://doi.org/10.1145/2737924.2737978.

[62] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. Intermittent Computing: Challenges and Opportunities. pages 8:1–8:14, 2017. doi: 10.4230/LIPIcs.SNAPL.2017.8.

[63] P. Maene, J. Gotzfried, R. de Clercq, T. Muller, F. Freiling, and I. Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE*

Transactions on Computers, PP(99):1–1, 2017. ISSN 0018-9340. doi: 10.1109/TC.2017.2647955.

[64] Kiwan Maeng and Brandon Lucia. Supporting peripherals in intermittent systems with just-in-time checkpoints. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 1101–1116. ACM, 2019. doi: 10.1145/3314221.3314613. URL https://doi.org/10.1145/3314221.3314613.

[65] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent Execution Without Checkpoints. *Proc. ACM Program. Lang.*, 1(OOPSLA):96:1–96:30, October 2017. ISSN 2475-1421. doi: 10.1145/3133920. URL http://doi.acm.org/10.1145/3133920.

[66] Sparsh Mittal and Ahmed Izzat Alsalibi. A survey of techniques for improving security of non-volatile memories. *Journal of Hardware and Systems Security*, 2(2):179–200, Jun 2018. ISSN 2509-3436. doi: 10.1007/s41635-018-0034-5. URL https://doi.org/10.1007/s41635-018-0034-5.

[67] Hyo-Chang Nam, Jong Kim, Sung Je Hong, and Sunggu Lee. A secure checkpointing system. In *8th Pacific Rim International Symposium on Dependable Computing (PRDC 2001), 17-19 December 2001, Seoul, Korea*, pages 49–56, 2001. doi: 10.1109/PRDC.2001.992679. URL https://doi.org/10.1109/PRDC.2001.992679.

[68] C. Navarro, S. Navarro, C. Marquez, L. Donetti, C. Sampedro, S. Karg, H. Riel, and F. Gamiz. InGaAs capacitor-less DRAM cells TCAD demonstration. *IEEE Journal of the Electron Devices Society*, pages 1–1, 2018. doi: 10.1109/JEDS.2018.2859233.

[69] NIST. Lightweight Cryptography Competition, 2018. https://csrc.nist.gov/projects/lightweight-cryptography.

[70] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Trans. Priv. Secur.*, 20(3):7:1–7:33, July 2017. ISSN 2471-2566. doi: 10.1145/3079763. URL http://doi.acm.org/10.1145/3079763.

[71] Krishna Pabbuleti, Deepak Mane, and Patrick Schaumont. Energy Budget Analysis for Signature Protocols on a Self-powered Wireless Sensor Node. In Nitesh Saxena and Ahmad-Reza Sadeghi, editors, *Radio Frequency Identification: Security and Privacy Issues*, pages 123–136, Cham, 2014. Springer International Publishing. ISBN 978-3-319-13066-8.

[72] James Pallister, Simon J. Hollis, and Jeremy Bennett. BEEBS: open benchmarks for energy measurements on embedded platforms. *CoRR*, abs/1308.5174, 2013. URL http://arxiv.org/abs/1308.5174.

[73] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. pages 379–394, May 2011. doi: 10.1109/SP.2011.38.

[74] Jens F. Peters, Manuel Baumann, Benedikt Zimmermann, Jessica Braun, and Marcel Weil. The environmental impact of li-ion batteries and the role of key parameters â€“ a review. *Renewable and Sustainable Energy Reviews*, 67:491–506, 2017. ISSN 1364-0321. doi: https://doi.org/10.1016/j.rser.2016.08.039. URL https://www.sciencedirect.com/science/article/pii/S1364032116304713.

[75] David R. Piegdon. Hacking in physically addressable memory. In *Seminar of Advanced Exploitation Techniques*, WS 2006/2007, 2006.

[76] Amir Rahmati, Mastooreh Salajegheh, Daniel E. Holcomb, Jacob Sorber, Wayne P. Burleson, and Kevin Fu. TARDIS: Time and Remanence Decay in SRAM to Implement Secure Protocols on Embedded Devices without Clocks. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 221–236, 2012. URL https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/rahmati.

[77] S. R. Jino Ramson, D. Jackuline Moni, S. Vishnu, Theodoros Anagnostopoulos, A. Alfred Kirubaraj, and Xiaozhe Fan. An iot-based bin level monitoring system for solid waste management. *Journal of Material Cycles and Waste Management*, 2021. URL https://doi.org/10.1007/s10163-020-01137-9.

[78] Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: System support for long-running computation on rfid-scale devices. *SIGARCH Comput. Archit. News*, 39(1):159–170, March 2011. ISSN 0163-5964. doi: 10.1145/1961295.1950386. URL http://doi.acm.org/10.1145/1961295.1950386.

[79] Srivaths Ravi, Anand Raghunathan, Paul C. Kocher, and Sunil Hattangady. Security in embedded systems: Design challenges. *ACM Trans. Embed. Comput. Syst.*, 3(3): 461–491, 2004. doi: 10.1145/1015047.1015049. URL https://doi.org/10.1145/1015047.1015049.

[80] James Reed, Joshua Daniels, Ayaz Siddiqui, Mitchell Cobb, and Chris Vermillion. Optimal exploration and charging for an autonomous underwater vehicle with energy-harvesting kite. In *2020 American Control Conference (ACC)*, pages 4134–4139, 2020. doi: 10.23919/ACC45564.2020.9147746.

[81] Phillip Rogaway. Authenticated-encryption with associated-data. pages 98–107, 2002. doi: 10.1145/586110.586125.

[82] Sunanda Roy, Jun-Jiat Tiang, Mardeni Roslee, Md. Tanvir Ahmed, and M. A. Parvez Mahmud. A quad-band stacked hybrid ambient rf-solar energy harvester with higher rf-to-dc rectification efficiency. *IEEE Access*, 9:39303–39321, 2021. doi: 10.1109/ACCESS.2021.3064348. URL https://doi.org/10.1109/ACCESS.2021.3064348.

[83] Archanaa S. Krishnan and Patrick Schaumont. Exploiting security vulnerabilities in intermittent computing: 8th international conference, space 2018, kanpur, india, december 15-19, 2018, proceedings. pages 104–124, 12 2018. ISBN 978-3-030-05071-9. doi: 10.1007/978-3-030-05072-6_7.

[84] D. Samyde, S. Skorobogatov, R. Anderson, and J. J. Quisquater. On a new way to read data from memory. In *First International IEEE Security in Storage Workshop, 2002. Proceedings.*, pages 65–69, Dec 2002. doi: 10.1109/SISW.2002.1183512.

[85] M. Sharad, R. Venkatesan, A. Raghunathan, and K. Roy. Multi-level magnetic RAM using domain wall shift for energy-efficient, high-density caches. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 64–69, Sept 2013. doi: 10.1109/ISLPED.2013.6629268.

[86] SLAS826H. MSP432P401R,MSP432P401 MSimpleLink Mixed-SignalMicrocontrollers. Technical report, Texas Instruments, March 2015. Revised August 2018. Available at https://www.ti.com/lit/ds/symlink/msp432p401r.pdf.

[87] slase54c. MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers. Technical report, Texas Instruments, March 2016. Revised August 2018. Available at http://www.ti.com/lit/ds/slase54c/slase54c.pdf.

[88] slau678a. MSP430FR5994 LaunchPad™ Development Kit (MSP-EXP430FR5994). Technical report, Texas Instruments, March 2016. URL http://www.ti.com/lit/ug/slau678a/slau678a.pdf.

[89] Ingo Stark. Integrating Thermoelectric Technology into Clothing for Generating Usable Energy to Power Wireless Devices. In *Proceedings of the Conference on Wireless Health*, WH '12, pages 17:1–17:2, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1760-3. doi: 10.1145/2448096.2448113. URL http://doi.acm.org/10.1145/2448096.2448113.

[90] Raoul Strackx and Frank Piessens. Ariadne: A minimal approach to state continuity. pages 875–892, Austin, TX, 2016. USENIX Association. ISBN 978-1-931971-32-4.

[91] Raoul Strackx, Bart Jacobs, and Frank Piessens. ICE: A passive, high-speed, state-continuity scheme. ACSAC '14, pages 106–115, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3005-3. doi: 10.1145/2664243.2664259.

[92] DSF 3V Supercapacitor. Dsf447q3r0 datasheet. URL https://www.cde.com/resources/catalogs/DSF.pdf.

[93] Milijana Surbatovich, Limin Jia, and Brandon Lucia. I/O dependent idempotence bugs in intermittent systems. *Proc. ACM Program. Lang.*, 3(OOPSLA):183:1–183:31, 2019. doi: 10.1145/3360609. URL https://doi.org/10.1145/3360609.

[94] Shivam Swami and Kartik Mohanram. ACME: Advanced Counter Mode Encryption for Secure Non-volatile Memories. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 86:1–86:6, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5700-5. doi: 10.1145/3195970.3195983. URL http://doi.acm.org/10.1145/3195970.3195983.

[95] Avalanche Technology. 8Mbit – 64Mbit Embedded MRAM Macro (eMRAM), 2019. URL https://www.avalanche-technology.com/wp-content/uploads/2019/06/8Mb-64Mb-eMRAM.pdf.

[96] *MSP MCU FRAM Utilities.* Texas Instruments, 2017.

[97] *MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers.* Texas Instruments, 2017.

[98] M.V. Tholl, H.G. AkarÃ§ay, H. Tanner, T. Niederhauser, A. Zurbuchen, M. Frenz, and A. Haeberlin. Subdermal solar energy harvesting â€“ a new way to power autonomous electric implants. *Applied Energy*, 269:114948, 2020. ISSN 0306-2619. doi: https://doi.org/10.1016/j.apenergy.2020.114948. URL https://www.sciencedirect.com/science/article/pii/S0306261920304608.

[99] Evgeny Tsimbalo, Xenofon Fafoutis, and Robert J. Piechocki. Fix it, don't bin it! - CRC error correction in bluetooth low energy. In *2nd IEEE World Forum on Internet of Things, WF-IoT 2015, Milan, Italy, December 14-16, 2015*, pages 286–290. IEEE Computer Society, 2015. doi: 10.1109/WF-IoT.2015.7389067. URL https://doi.org/10.1109/WF-IoT.2015.7389067.

[100] Emanuele Valea, Mathieu Da Silva, Giorgio Di Natale, Marie-Lise Flottes, Sophie Dupuis, and Bruno Rouzeyre. SECCS: secure context saving for iot devices. *CoRR*, abs/1903.04314, 2019. URL http://arxiv.org/abs/1903.04314.

[101] Joel Van Der Woude and Matthew Hicks. Intermittent Computation Without Hardware Support or Programmer Intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 17–32, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL http://dl.acm.org/citation.cfm?id=3026877.3026880.

[102] Harrison Williams, Xun Jian, and Matthew Hicks. Forget failure: Exploiting SRAM data remanence for low-overhead intermittent computation. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 69–84. ACM, 2020. doi: 10.1145/3373376.3378478. URL https://doi.org/10.1145/3373376.3378478.

[103] C. F. Yang, K. H. Chen, Y. C. Chen, and T. C. Chang. Fabrication of one-transistor-capacitor structure of nonvolatile TFT Ferroelectric RAM devices using ba(zr0.1ti0.9)o3 gated oxide film. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 54(9):1726–1730, September 2007. ISSN 0885-3010. doi: 10.1109/TUFFC.2007.457.

[104] So-Nam Yun, Young-Bog Ham, and J. H. Park. Energy harvester using PZT actuator with a cantilver. In *2009 ICCAS-SICE*, pages 5514–5517, Aug 2009.