Benchmarking and Configuring Security Levels in Intermittent Computing

ARCHANAA S. KRISHNAN, Virginia Tech, USA
PATRICK SCHAUMONT, Worcester Polytechnic University, USA

Intermittent computing derives its name from the intermittent character of the power source used to drive the computing, typically an energy harvester of ambient energy sources. Intermittent computing is characterized by frequent transitions between the powered and the non-powered state. To enable the processor to quickly recover from unexpected power loss, regular checkpoints store the run-time state of the program, including variables, control information, and machine state. In sensitive applications such as logged measurements, checkpoints must be secured against tamper and replay. We investigate the overhead of creating, securing, and restoring checkpoints with respect to the application. We propose a configurable checkpoint security setting that leverages application properties to reduce overhead of checkpoint security and implement the same using a secure checkpointing protocol. We discuss a prototype implementation for a FRAM-based microcontroller, and we characterize the cost of adding and configuring security to traditional checkpointing using a suite of embedded benchmark applications.

CCS Concepts: • **Security and privacy → Embedded systems security**; *Access control;*

Additional Key Words and Phrases: Intermittent computing, checkpoint security, non-volatile memory, embedded systems, benchmark, AEAD

ACM Reference format:
Archanaa S. Krishnan and Patrick Schaumont. 2022. Benchmarking and Configuring Security Levels in Intermittent Computing. *ACM Trans. Embedd. Comput. Syst.* 21, 4, Article 36 (September 2022), 22 pages.
https://doi.org/10.1145/3522748

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 grid, 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 [34]. The quantity of devices, on the one hand, and the level of trust placed in them, on the other hand, has important implications for the realization of IoT devices.

This work was supported in part by NSF Grant No. 1704176.
Authors' addresses: A. S. Krishnan, Virginia Tech, Blacksburg, Virginia, USA; email: archanaa@vt.edu; P. Schaumont, Worcester Polytechnic University, Worcester, Massachusetts, USA; email: pschaumont@wpi.edu.

1539-9087/2022/09-ART36
https://doi.org/10.1145/3522748

First, powering the burgeoning number of devices becomes a major challenge. 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 [33], wind [21], RF [33], and vibration [22]. By means of 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 mW 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 autonomous operation of IoT devices, they do not guarantee continuous operation of the IoT device for two reasons. First, the source of ambient energy itself may be discontinuous. Solar cells do not deliver power at night, and vibration energy harvesters do not 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 [30]. 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 [23]. The creation and restoration of a checkpoint requires energy and clock cycles, which impacts 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 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 [14, 24, 36, 40], continuity of data flow [16, 24], retention of peripheral state [3, 7, 8, 26], processing time sensitive data [18, 36], and optimizing checkpoint size [2]. While the above features ensure 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 in a correct and secure manner. 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 [31]. A broad class of cryptographic algorithms and dedicated security protocols provide the tools and mechanisms to build trust [17]. In addition, security architectures ensure that these cryptographic algorithms themselves operate as expected, free from tampering and malicious influence [11, 25, 28]. 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 [19], 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 [4, 9] and cryptographic primitives [4, 12, 20, 39]. 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 [4, 12, 20, 39]. In particular, we investigate and benchmark how application-level security concerns map into the security primitives developed for secure checkpointing.

Our work resides at the crossing of intermittent computing and the security challenges required for a secure IoT. 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 also investigate the state-of-the-art secure intermittent computing solutions to compare its security policy and their effects on the application. To that end, we identified that all the solutions use a one-size-fits-all security policy for the entire checkpoint as they are agnostic to the application-level security requirements. This may hinder forward progress of the application as applying certain security properties to the entire checkpoint consumes energy that may otherwise be used by the application. We propose to consider the needs of the application in deciding the security properties required by the intermittent computing solution. To the best of our knowledge, there is no prior work in secure intermittent computing that has considered the interplay of application level security needs and application efficiency in intermittent settings. The key contributions of our work are as follows:

- We analyse the role of application in the overhead of intermittent computing and its security using a curated list of IoT benchmark applications.
- 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 [20] and incorporate the proposed configurable checkpoint security in its implementation. We evaluate the implementation with our benchmarks.
- We will provide the source code for our experiments, including benchmarks, optimized checkpoint security solution, and our configurable checkpoint security levels, upon publication.[1]

Organization. In Section 2, we provide a brief overview of intermittent computing, its security requirements, and the state-of-the-art checkpoint security solutions before discussing the effects of applications on checkpoints using a set of benchmarks in Section 3. In Section 4, we propose a configurable checkpoint security setting that leverages the application to reduce overhead of securing checkpoints. In Section 5, we describe our implementation of the configurable checkpoint

[1] https://github.com/Secure-Embedded-Systems/benchmark-SLx-IC.

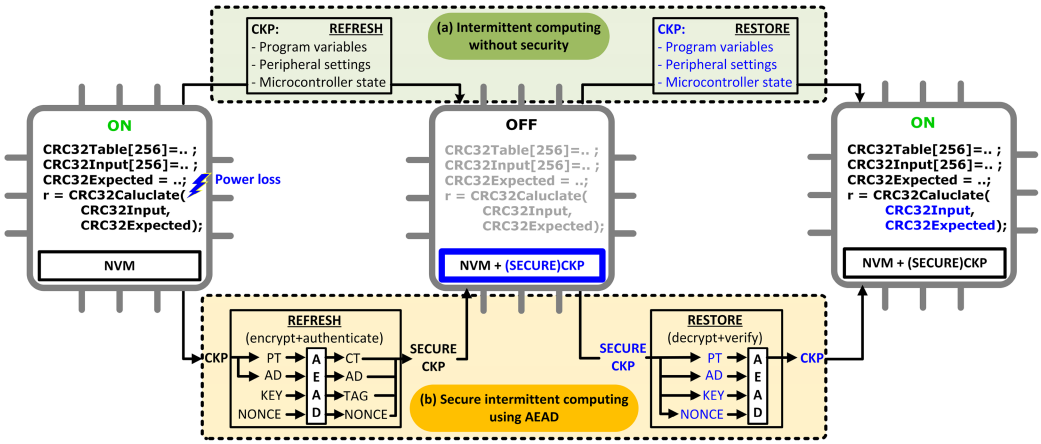


Fig. 1. CRC32 verification as an example intermittent application running on a microcontroller with non-volatile memory (NVM). (a) Unsecure intermittent computing that stores plaintext checkpoints (CKP) and restores checkpoints without any security checks. (b) Secure intermittent computing using AEAD to encode security properties such integrity, authenticity, confidentiality, and freshness into secure checkpoints (SECURE CKP), which are verified before restoring the decoded checkpoint (CKP).

security using a secure checkpoint protocol and its evaluation on our benchmarks, followed by conclusions in Section 6.

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 [38] and IEE 802.15.4 [15], 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 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 necessary state required for forward progress of the application using *refresh* operation. In the top half of Figure 1, the **checkpoint (CKP)** contains program variables, peripheral settings, and micrcontroller state. We elaborate on the contents of a checkpoint in Section 3.2.1. 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.

Table 1. Checkpoint Security Properties Satisfied by the State-of-the-art Related Work

Checkpoint security properties	Ghodsi et al. [12]	SECCS [39]	Asad et al. [4]	SICP [20]
Integrity & Authenticity	–	✓	✓	✓
Freshness	–	–	–	✓
Availability	–	–	–	✓
Confidentiality	✓	✓	✓	✓

2.1 Attacker Model

We assume an attacker that aims to gain access to data from checkpoints of an intermittent system. To that end, the attacker has two capabilities. First, the attacker can control the power supplied to the device, for example, by tampering with the energy harvesting circuitry. The attacker can arbitrarily start and stop the device to gain useful information from checkpoint. The scope of the attacker is not to disrupt the forward progress of the application, for example, by cutting-of power supplied to the energy-harvested device. Second, the attacker can read from and write to sections of the non-volatile memory that are not protected. We assume a small section of non-volatile memory is protected from the attacker to store data that needs protection from unauthorized read and write access. We cannot place the entire checkpoint in this tamper-free memory, because the size of tamper-free memory depends on the device and the size of checkpoint depends on the application. To ensure a generic attacker model and secure checkpointing solution, we assume only a small part of the checkpoint is placed in the tamper-free non-volatile memory. We assume that the cryptographic keys used by the secure intermittent solution are protected from unauthorized access from the attacker, which may be achieved by storing the keys in the tamper-free non-volatile memory to protect them from our attacker.

2.2 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 1. While this is a stateful computation model, it does not guarantee statefulness 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, then they can snoop, spoof, and replay checkpoints [19]. 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 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, freshness of the checkpoint must be guaranteed to prevent 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 lack of a valid checkpoint. Finally, the checkpoint may require confidentiality guarantees based on the contents that require protection from unauthorized access.

Related work. Table 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. [12] only encrypt the

checkpoint without considering the other requirements. **SECure Context Saving (SECCS)** [39] and Asad et al. [4] ensure the information security of checkpoint, including confidentiality, integrity and authenticity, using encryption and authentication algorithms such as **Authenticated Encryption with Associated Data (AEAD)** [32]. The **Secure Intermittent Computing Protocol (SICP)** [20] satisfies all the minimum security requirements. SICP also uses AEAD to ensure information security properties of the checkpoint and its freshness. SICP is the only solution that ensures 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 [4, 20, 39] use AEAD or a combination of encryption and authentication to protect checkpoints. We capitalize on the versatility of AEAD in Section 4 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 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.

3 ROLE OF APPLICATION IN CHECKPOINTS

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 our work, we focus on how the contents of the checkpoint is partially dependent on the application, and we leverage this dependency to propose configurable checkpoint security 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.

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 ultra-low power microcontroller that only consumes 120 μA/MHz of active current [1]. It is equipped with 256 kB of Ferroelectric RAM (FRAM) and 8 kB 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:

$$E = \frac{V_{CC}}{R} \int_{t_1}^{t_2} v(t)\, dt. \tag{1}$$

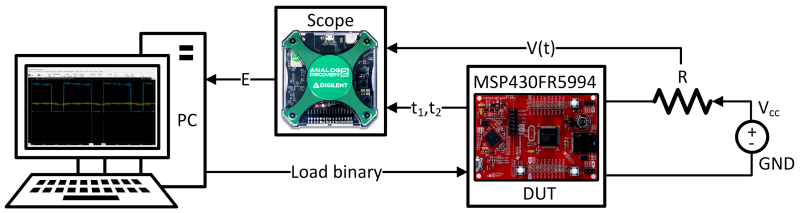


Fig. 2. Block diagram of the experimental setup. PC is used to load benchmark binaries onto MSP430FR5994 LaunchPad Development Kit, which is powered using a 3.5 V (V_{CC}) DC power supply via a 1 kΩ(R) shunt resistor. A Digilent Anolog Discovery 2 USB Oscilloscope, which is triggered using the GPIO pins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

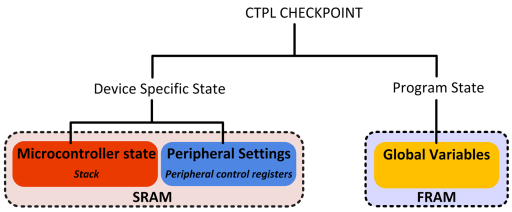
Experimental setup. The measurements were collected on MSP430FR5994 LaunchPad Development Kit across a 1 kΩ shunt resistor using Digilent Anolog Discovery 2 USB Oscilloscope. The scope was operated at 1MHz and triggered using on-chip GPIO to identify measurements for target functions. The microcontroller was powered using an external DC power supply at V_{CC} = 3.5 V, as illustrated in Figure 2, and operated at 8 MHz using on-chip clock source. The energy consumption of a function's execution is computed using Equation (1), 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 [37]. 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, then the checkpoint, which contains 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.

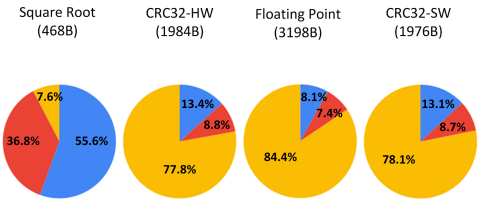
3.2 Benchmark Applications

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 2 from BEEBS [29] benchmark suite. 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



(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.

Fig. 3. Similarities in checkpoint partitions and differences in checkpoint content across benchmarks.

Table 2. Variation in Benchmark Cost, Checkpoint Size, and Contents and the Similarities in CTPL Overhead and Device-specific State Across Different Benchmark Applications

	Benchmark cost		CTPL overhead		Checkpoint size		
Benchmark	**Energy (μJ)**	**Time (ms)**	**Energy (μJ)**	**Time (ms)**	**Program variables (B)**	**Device-specific state(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	2,416	432	2,848
Floating Point	6,986.00	2,605.00	1.3	0.5	2,700	498	3,198
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	1,544	432	1,976
CRC32-HW	1.38	0.49	2.5	0.9	1,810	174	1,984

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.

contents across benchmarks, which provides a foundation for configuring checkpoint security based on the needs of application.

3.2.1 *Similarities Among Benchmarks.*

Checkpoint partition. We broadly partition the contents of checkpoints in all benchmarks into device-specific state and program variables. Figure 3(a) 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 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. The global variables include the inputs, outputs, keys, tables, and static constants used by the application. By placing checkpoint generation calls after benchmark functions, the

local variables used by the benchmark functions are not a part of application stack. Thus, we only checkpoint global variables. The choice of adding global and/or local variables depends on the choice of intermittent computing technique and the frequency of checkpoint.

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 regular operation of 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, as discussed in the differences among benchmarks in Section 3.2.2.

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 contain checkpoint data required for restoring the microcontroller, peripherals and a few volatile application variables. Table 2 lists the size of device-specific state for all benchmarks and energy and time required to create and restore a checkpoint of device-specific state under CTPL overhead. The measurements were computed by placing checkpoint calls at 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 is attributed to the similarity in device-specific state sizes. In our experiments, on average, the checkpoint generation and restoration for benchmarks consumed 2.2 mJ of energy and introduced 0.8 s latency.

3.2.2 *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 CRC32 peripheral used in CRC32-HW benchmark, which adds an additional 6B to the total checkpoint size when compared to CRC32-SW benchmark. Figure 3(b) illustrates the variation in checkpoint content for a few selected benchmarks using the data provided in Table 2. The peripheral settings contributes to the majority of checkpoint content in 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 2 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 contains 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 2 lists 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 program variable as the CRC32 peripheral processes inputs from the benchmark. This reduces the device-specific 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 2 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 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 40× improvement. The variation in the performance overhead of each benchmark demonstrates the change in energy requirement for each applications, which is elaborated in Section 5.5.

4 CONFIGURABLE MULTI-LEVEL CHECKPOINT SECURITY

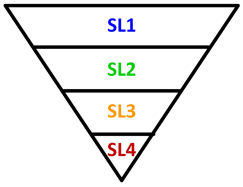
A checkpoint contains a snapshot of all the data necessary to resume the progress of the application. As described in the previous section, the contents of the checkpoint are largely dependent on the application. Let us consider the CRC32-HW benchmark. Apart from device-specific data such as the stack 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 [39] to secure their checkpoints, then the entire checkpoint will be encrypted and authenticated. Similar to SECCS, the other solutions listed in Table 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 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 1. We also propose certain optimizations specific to the selected solution to minimize the overhead from securing checkpoints.

4.1 Multi-level Checkpoint Security

Our multi-level checkpoint security involves four levels, illustrated in Figure 4. We leverage the design of AEAD described in Section 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).

4.1.1 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



	Security Properties				Overhead	Checkpoint division	
	Confidentiality	Integrity & Authenticity	Freshness	Availability		Plaintext	Associated data
SL1	●	●	●	●	++++	●	-
SL2	◗	●	●	●	+++	◗	◗
SL3	-	●	●	●	++	-	●
SL4	-	-	-	-	+	-	-

Fig. 4. 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.

cryptography is involved in the encoding of a checkpoint. SL4 is equivalent to unsecured intermittent computing systems.

4.1.2 SL3: No Confidentiality. If an application does not generate checkpoints with sensitive content that require confidentiality, then SL3 is sufficient to ensure forward progress of the application 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 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.

4.1.3 SL2: Partial Confidentiality. A few applications may contain sensitive data of long running application 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 portioning the checkpoint into secure and non-secure sections. For example, we may consider all the program variables in Table 2 to be secure and the device-specific state to be non-secure. Both the secure and non-secure section requires SL3 level security guarantees, whereas, the secure section also requires confidentiality guarantees. The secure section is input as plaintext to AEAD and the non-secure section is used as associated data.

4.1.4 SL1: Full Confidentiality. SL1 guarantees 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 1 that employ one-size-fits-all security policy to the entire checkpoint.

4.2 Configuring and Optimizing Checkpoint Security Using SICP

4.2.1 Selecting A Checkpoint Security Solution. We chose the SICP [20] 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 1 that ensures 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

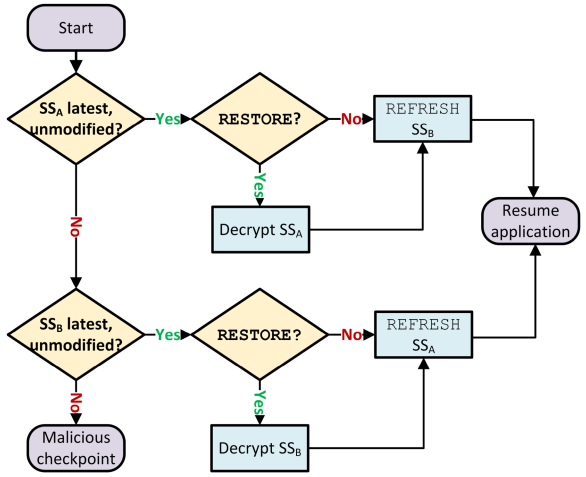


Fig. 5. A flow chart of the original SICP algorithm REFRESH and RESTORE using state save packet, SS_i, to store secure checkpoints in alternating buffers A and B. Both the algorithms detect the latest unmodified buffer cryptographically using AEAD. RESTORE creates a new state save packet with a new nonce and the latest checkpoint without any forward progress in the application.

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 checkpoint, as discussed in Section 4.1.

4.2.2 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 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 checkpoints. While this maybe a potential checkpoint security solution, it is not applicable for all benchmarks and devices. The size of checkpoint varies based on the benchmark, as listed in Table 2, 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 previous checkpoint is used as associated data input for the latest checkpoint to ensure only one of the buffer contains a valid checkpoint. Figure 5 illustrates the flow of generation and restoration of a secure checkpoint, also known as state save packet, SS, with encoded security properties, which contains the encrypted checkpoint, authentication tag, and the nonce. Both REFRESH and RESTORE identifies the latest checkpoint between A and B by first verifying integrity and authenticity of checkpoint A. If buffer A is the latest unmodified checkpoint, then 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, then buffer B is checked for the same and buffer A is updated.

Table 3. 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

	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

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 *Pub* and *Pri* to apply confidentiality properties to section PRI.

Algorithm 1 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 availability of a secure checkpoint even if the latest checkpoint is incomplete. Apart from `REFRESH` and `RESTORE` described in Algorithm 1, 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 [20].

4.2.3 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 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, then they will apply the same security properties across the entire checkpoint. Whereas, selecting SL2 involves partitioning the checkpoint into secure, *Pri*, and non-secure sections, *Pub*, as described under section mapping in Table 3.

Now, the checkpoint is divided into non-secure section, *Pub*, which requires integrity, authenticity, and freshness, and secure 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 secure section of checkpoint. The rest of the checkpoint, which is in the non-secure section, is provided to AEAD as associated data along with the authentication tag from previous checkpoint, as in original SICP. Table 3 states the one to one mapping between associated data and non-secure section, and plaintext and secure 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 1. Table 3 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 checkpoint is considered non-secure and passed as associated data. In SL1, the entire checkpoint is in *Pri* and provided as plaintext input to AEAD

ALGORITHM 1: 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 ← 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 ← 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 ← Q
10:         S_B ← AEAD_encr(STATE, T_A|NS_B, R_B, K)
11:         T_B ← AEAD_auth(S_B, T_A|NS_B, R_B, K)
12:         f_A, f_B ← (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 ← 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 ← Q
22:         S_A ← AEAD_encr(STATE, T_B|NS_A, R_A, K)
23:         T_A ← AEAD_auth(S_A, T_B|NS_A, R_A, K)
24:         f_B, f_A ← (0, 1)
25:     end if
26: else
27:     abort()
28: end if
```

as in original SICP. In SL2, *Pri* and *Pub* are inputs for plaintext and associated data, respectively, in AEAD operations.

4.2.4 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 1.

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 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 1. 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 set flag to either restore the decrypted checkpoint if it passes verification check or update the other checkpoint buffer with latest checkpoint. This flag check is added on lines 3 and 15 in Algorithm 1.

5 IMPLEMENTATION

In this section, we present a detailed overview of implementing our proposed configurable mulit-level checkpoint security using SICP. We utilise MSP430FR5994, described in Section 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. We

Table 4. 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

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

The overhead listed is measured for encrypting and authenticating 16B each of plaintext and associated data using 16B key and nonce.

provide a brief description for developers to use multi-level security in their secure energy harvesting system.

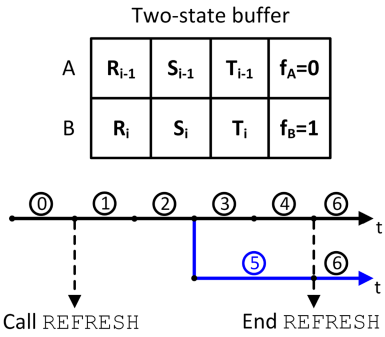
5.1 Cryptographic Primitive

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[27]. In Table 4, we present performance overhead of two selected ciphers, Ascon [10] and GIFT-COFB [5]. Ascon was the only cipher with 16-bit optimized submission that 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 optimised or 64-bit optimized implementations. We present results of the reference implementation provided with the GIFT-COFB submission. Next, we also selected EAX [6] as a representative of hardware accelerated AEAD schemes. Our target device is equipped with AES256 accelerator for encryption and decryption.

Table 4 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 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 that we chose an AEAD scheme with minimal overhead in both energy and time. As expected, from Table 4, 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.

5.2 Optimized SICP Implementation

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, to generate secure checkpoints, to wipe secure state, and to restore unmodified secure checkpoints. SICP uses a 128-bit counter initialized to a random number as 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



	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

Fig. 6. 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 device loses power.

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 [13]. 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.

Atomic operations. Since our input source is intermittent, we must ensure that certain memory writes are performed atomically. Figure 6 illustrates the need for atomic update of the authentication tag and flags in steps 11, 12 and 23, 24 in Algorithm 1. 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 size. But the size and availability of residual capacitance is platform dependent. For example, MSP430FR5994 LaunchPad Development Kit contains one 10 μF capacitor and three 100 nF capacitors, which provides sufficient residual energy to consistently overwrite up to 16 kB of memory after detecting a power loss. To provide a generic implementation, we use a two-state secure checkpoint buffer and implement availability feature with atomic writes using residual capacitance. 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

Table 5. Performance Overhead of Generating and Restoring Checkpoints Securely Using Multi-level Security-based Optimized SICP Implementing in Various Benchmarks

Benchmark	SL4		SL3		SL2		SL1	
	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	1,014.0	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

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.

5.3 Multi-level Checkpoint Security

We use MSP430FR5994's linker description file to define two new sections of non-volatile memory. First, we define the secure section, `.private`, from 0x10000 to 0x10FFF. Second, we define the non-secure section, `.public`, from 0x11000 to 0x11FFF. We chose 4 kB 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 3,198 bytes, which fits in 4 kB 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 secure memory, as illustrated using the following code snippet from floating point benchmark. All the other checkpoint data is placed in non-secure memory using `__attribute__((section (".public")))`. For ease-of-use, we define preprocessor directives, such as `PUBLIC_BENCH` to place each part of checkpoint, including the program variables, micrcontroller data, and peripheral data, in either secure or non-secure memory 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 secure and non-secure sections, which provides the size of plaintext and associated data provided to AEAD. We then provide the plaintext (S) and associated data (NS) as input to $AEAD_{encr}()$ and $AEAD_{auth}()$. The size of each memory section varies depending on the security level and the size of checkpoint, as described in Table 3. From Table 2, we can see the varying checkpoint memory requirement of each benchmark. In all our implementations, SL1 and SL3 contains the entire checkpoint in either secure or non-secure state. For simplicity, in SL2, the secure state contains all program variables and the non-secure

state contains the device-specific state. We arrived at this partition, because we aim to protect the confidentiality of application state from the attacker whereas the device-specific state does not necessarily need confidentiality protection. In all implementations but CRC32-HW, we place the the peripheral registers as a part of device-specific state in non-secure memory in SL2. For CRC32-HW, we place the peripheral registers as a part of program variables.

5.4 Developer's Guide

We provide a brief overview of the steps required to implement secure checkpoints with multi-level security. We assume that the developer already has the necessary hardware including a target device with non-volatile memory and energy harvesting circuit. First, the developer needs to select their choice of intermittent computing technique, which provides the API for checkpoint generation. Second, the developer needs to select their choice of secure intermittent computing solution, which provides the mechanism for secure checkpoint generation. The developer can then identify device-specific and application-specific state based on the intermittent computing, as we described in Section 2. The partition of checkpoint between public and private is at the discretion of the developer and the needs of the application. Third, the developer can create dedicated memory sections for storing public and private sections of checkpoint. Once the developer decides the security properties of checkpoint content, they can place the content in dedicated section as described in Section 5.3. The developer then needs to make modifications to the secure intermittent computing solution to selectively apply security policies for public and private sections. Finally, depending on the choice of intermittent computing technique, the developer must either place secure checkpoint generation and restoration calls in the application or the intermittent computing technique automatically generates and restore secure checkpoints on detecting power loss.

5.5 Results

The measurements reported in this section were measured on the same experimental setup described in Section 3.1. We use energy and time as metrics to evaluate 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. Table 5 lists the energy and time overhead of securely generating and restoring one checkpoint using SL4, SL3, SL2, and SL1. The overhead of checkpointing at SL4 is also listed in Table 2, as SL4 is equivalent to unsecure intermittent computing. The overhead listed for each level includes secure checkpoint generation and restoration.

5.5.1 Improvements with Multi-level Security. Figure 7 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 2 as baseline energy consumption. A similar trend in 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 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 1,000B). 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 overhead of checkpoint security is vastly dependent on the application and the programmer needs to carefully select security level based on the contents of the checkpoint to avoid unnecessary overhead incurred from encryption/decryption.

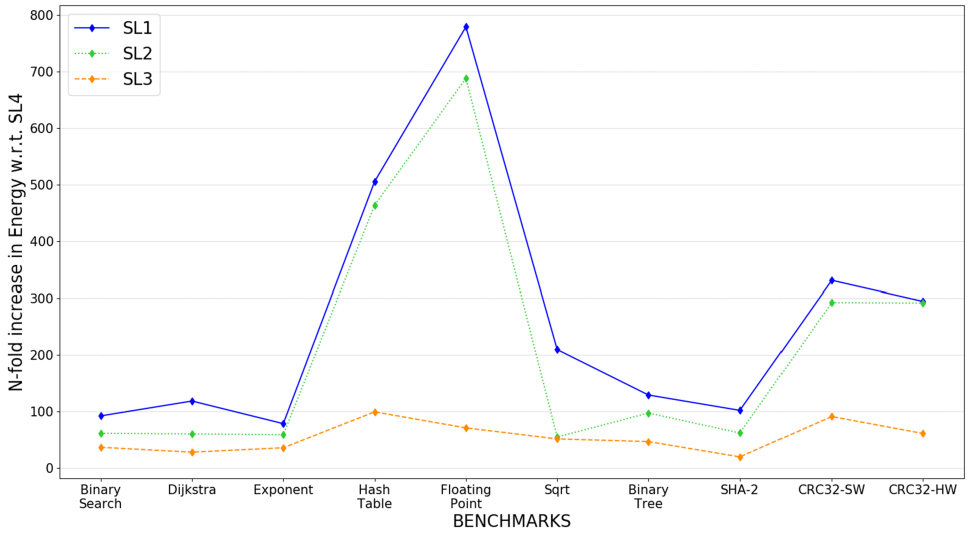


Fig. 7. N-fold increase in energy required to secure checkpoints of various benchmarks when operated at different security levels (SL1-3) with respect to (w.r.t.) the energy required for unsecure checkpoints (SL4).

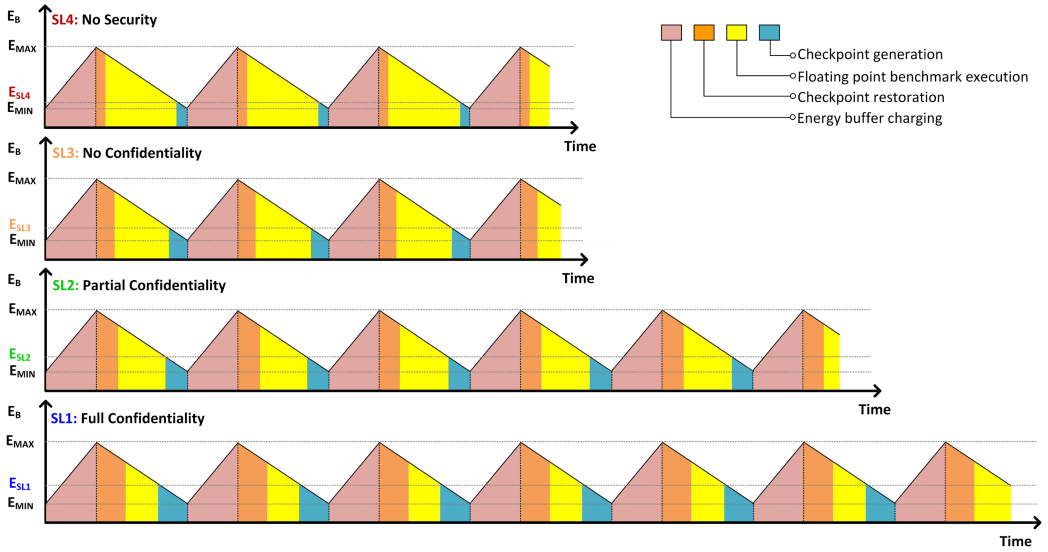


Fig. 8. A hypothetical power cycle graph of an ideal 470 μF supercapacitor buffer used to complete one iteration of the floating point arithmetic benchmark. E_{max} = 2.1 mJ, E_{SL4} = 1.3 μJ, E_{SL3} = 0.09 mJ, E_{SL2} = 0.9 mJ, and E_{SL1} = 1.01 mJ. With the increase in security level from SL4 to SL1, the amount of energy available for forward progress of the benchmark during each power cycle is reduced to accommodate securely refreshing and restoring checkpoints, which in-turn increases the number 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 energy harvester.

5.5.2 Number of Checkpoints. The number of checkpoints required for each benchmark varies depending on the size of energy buffer, input power, security level, and benchmark itself. We hypothesize the charge-discharge cycles of an ideal 470 μF supercapacitor in Figure 8 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 [35], then it can provide 2.1 mJ of energy in one power cycle when the input voltage to the supercapacitor is 3 V. 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 incurs 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 8 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 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 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 energy harvester.

6 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. We demonstrate the need for customized checkpoint security solutions that is not available in the state-of-the-art secure intermittent computing solutions using benchmark applications. 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 secure and non-secure sections to avoid unnecessary encryption/decryption of non-secure section of the checkpoint, while ensuring other checkpoint security properties are still fulfilled. We provide a proof-of-concept implementation of our multi-level security using a secure checkpointing protocol on a low-power microcontroller. Based on our results, we conclude that the application plays a vital role in deciding, both the security properties of the checkpoints and the overhead of secure intermittent computing. In this work, we performed a coarse-grained partition of checkpoint 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.

REFERENCES

[1] Texas Instruments. 2016. MSP430FR5994 LaunchPad Development Kit (MSP-EXP430FR5994). Technical Report. Texas Instruments. Retrieved from http://www.ti.com/lit/ug/slau678a/slau678a.pdf.

[2] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2019. Efficient intermittent computing with differential checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'19)*, Jian-Jia Chen and Aviral Shrivastava (Eds.). ACM, 70–81. https://doi.org/10.1145/3316482.3326357

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

[4] Hafiz Areeb Asad, Erik Henricus Wouters, Naveed Anwar Bhatti, Luca Mottola, and Thiemo Voigt. 2020. On securing persistent state in intermittent computing. In *Proceedings of the 8th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems (ENSsys@SenSys'20)*. ACM, 8–14. https://doi.org/10.1145/3417308.3430267

[5] Subhadeep Banik, Avik Chakraborti, Tetsu Iwata, Kazuhiko Minematsu, Mridul Nandi, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. 2019. GIFT-COFB v1.0. Submission to Round 2 of the NIST Lightweight Cryptography project. Retrieved from https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/gift-cofb-spec-round2.pdf.

[6] Mihir Bellare, Phillip Rogaway, and David A. Wagner. 2004. The EAX mode of operation. In *Proceedings of the 11th International Workshop on Fast Software Encryption (FSE'04) (Lecture Notes in Computer Science, Vol. 3017)*, Bimal K. Roy and Willi Meier (Eds.). Springer, 389–407. https://doi.org/10.1007/978-3-540-25937-4_25

[7] Gautier Berthou, Pierre-Évariste Dagand, Delphine Demange, Rémi Oudin, and Tanguy Risset. 2020. Intermittent computing with peripherals, formally verified. In *Proceedings of the 21st ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'20)*, Jingling Xue and Changhee Jung (Eds.). ACM, 85–96. https://doi.org/10.1145/3372799.3394365

[8] Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. 2017. Peripheral state persistence for transiently-powered systems. In *Proceedings of the Global Internet of Things Summit (GIoTS'17)*. IEEE, 1–6. https://doi.org/10.1109/GIOTS.2017.8016243

[9] Daniel Dinu, Archanaa S. Krishnan, and Patrick Schaumont. 2019. SIA: Secure intermittent architecture for off-the-shelf resource-constrained microcontrollers. In *Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST'19)*. IEEE, 208–217. https://doi.org/10.1109/HST.2019.8740834

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

[11] Karim Eldefrawy, Aurélien Francillon, Daniele Perito, and Gene Tsudik. 2012. SMART: Secure and minimal architecture for (establishing a dynamic) root of trust. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS'12)*.

[12] Zahra Ghodsi, Siddharth Garg, and Ramesh Karri. 2017. Optimal checkpointing for secure intermittently-powered IoT devices. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'17)*, Sri Parameswaran (Ed.). IEEE, 376–383. https://doi.org/10.1109/ICCAD.2017.8203802

[13] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. 2008. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the 17th USENIX Security Symposium*, Paul C. van Oorschot (Ed.). USENIX Association, 45–60. Retrieved from http://www.usenix.org/events/sec08/tech/full_papers/halderman/halderman.pdf.

[14] Matthew Hicks. 2017. Clank: Architectural support for intermittent computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. ACM, 228–240. https://doi.org/10.1145/3079856.3080238

[15] IEEE. [n.d.]. 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). Retrieevd from https://standards.ieee.org/standard/802_15_4-2003.html.

[16] Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee, and Vijay Raghunathan. 2015. QuickRecall: A HW/SW approach for computing across power cycles in transiently powered computers. *ACM J. Emerg. Technol. Comput. Syst.* 12, 1 (2015), 8:1–8:19. https://doi.org/10.1145/2700249

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

[18] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah D. Hester, and Przemyslaw Pawelczak. 2020. Time-sensitive intermittent computing meets legacy software. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 85–99. https://doi.org/10.1145/3373376.3378476

[19] Archanaa S. Krishnan and Patrick Schaumont. 2018. Exploiting security vulnerabilities in intermittent computing. In *Proceedings of the 8th International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE'18) (Lecture Notes in Computer Science, Vol. 11348)*, Anupam Chattopadhyay, Chester Rebeiro, and Yuval Yarom (Eds.). Springer, 104–124. https://doi.org/10.1007/978-3-030-05072-6_7

[20] Archanaa S. Krishnan, Charles Suslowicz, Daniel Dinu, and Patrick Schaumont. 2019. Secure intermittent computing protocol: Protecting state across power loss. In *Proceedings of the Design, Automation & Test in Europe Conference*

& Exhibition (DATE'19), Jürgen Teich and Franco Fummi (Eds.). IEEE, 734–739. https://doi.org/10.23919/DATE.2019.8714997

[21] Xia Li, Zhiyuan Li, Cheng Bi, Benxue Liu, and Yufeng Su. 2020. Study on wind energy harvesting effect of a vehicle-mounted piezo-electromagnetic hybrid energy harvester. *IEEE Access* 8 (2020), 167631–167646. https://doi.org/10.1109/ACCESS.2020.3023649

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

[23] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. 2017. Intermittent computing: Challenges and opportunities. In *Proceedings of the 2nd Summit on Advances in Programming Languages (SNAPL'17) (LIPIcs, Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 8:1–8:14. https://doi.org/10.4230/LIPIcs.SNAPL.2017.8

[24] Brandon Lucia and Benjamin Ransford. 2015. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, David Grove and Steve Blackburn (Eds.). ACM, 575–585. https://doi.org/10.1145/2737924.2737978

[25] P. Maene, J. Gotzfried, R. de Clercq, T. Muller, F. Freiling, and I. Verbauwhede. 2017. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Trans. Comput.* 99 (2017), 1. https://doi.org/10.1109/TC.2017.2647955

[26] Kiwan Maeng and Brandon Lucia. 2019. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 1101–1116. https://doi.org/10.1145/3314221.3314613

[27] NIST. 2018. Lightweight Cryptography Competition. Retrieved from https://csrc.nist.gov/projects/lightweight-cryptography.

[28] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. 2017. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Trans. Priv. Secur.* 20, 3, Article 7 (July 2017), 33 pages. https://doi.org/10.1145/3079763

[29] James Pallister, Simon J. Hollis, and Jeremy Bennett. 2013. BEEBS: Open benchmarks for energy measurements on embedded platforms. Retrieved from http://arxiv.org/abs/1308.5174.

[30] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System support for long-running computation on RFID-scale devices. *SIGARCH Comput. Archit. News* 39, 1 (Mar. 2011), 159–170. https://doi.org/10.1145/1961295.1950386

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

[32] Phillip Rogaway. 2002. Authenticated-encryption with associated-data. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS'02)*, Vijayalakshmi Atluri (Ed.). ACM, 98–107. https://doi.org/10.1145/586110.586125

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

[34] Statista. 2020. Internet of Things (IoT Statistics Report). Retrieved from https://www.statista.com/study/27915/internet-of-things-iot-statista-dossier/.

[35] DSF 3V Supercapacitor. [n.d.]. DSF447Q3R0 Datasheet. Retrieved from https://www.cde.com/resources/catalogs/DSF.pdf.

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

[37] Texas Instruments. 2017. MSP MCU FRAM Utilities. Texas Instruments. Retrieved from https://www.ti.com/tool/MSP-FRAM-UTILITIES.

[38] Evgeny Tsimbalo, Xenofon Fafoutis, and Robert J. Piechocki. 2015. Fix it, don't bin it!—CRC error correction in bluetooth low energy. In *Proceedings of the 2nd IEEE World Forum on Internet of Things (WF-IoT'15)*. IEEE Computer Society, 286–290. https://doi.org/10.1109/WF-IoT.2015.7389067

[39] Emanuele Valea, Mathieu Da Silva, Giorgio Di Natale, Marie-Lise Flottes, Sophie Dupuis, and Bruno Rouzeyre. 2018. SECCS: SECure context saving for IoT devices. Retrieved from http://arxiv.org/abs/1903.04314.

[40] Harrison Williams, Xun Jian, and Matthew Hicks. 2020. Forget failure: Exploiting SRAM data remanence for low-overhead intermittent computation. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 69–84. https://doi.org/10.1145/3373376.3378478

Received June 2021; revised January 2022; accepted February 2022