Secure Intermittent Computing Protocol: Protecting State Across Power Loss

Archanaa S. Krishnan
Virginia Tech
Blacksburg, VA, USA
archanaa@vt.edu

Charles Suslowicz
Virginia Tech
Blacksburg, VA, USA
cesuslow@vt.edu

Daniel Dinu
Virginia Tech
Blacksburg, VA, USA
ddinu@vt.edu

Patrick Schaumont
Virginia Tech
Blacksburg, VA, USA
schaum@vt.edu

***Abstract*—Intermittent computing systems execute long-running tasks under a transient power supply such as an energy harvesting power source. During a power loss, they save intermediate program state as a checkpoint into write-efficient non-volatile memory. When the power is restored, the system state is reconstructed from the checkpoint, and the long-running computation continues. We analyze the security risks when power interruption is used as an attack vector, and we demonstrate the need to protect the integrity, authenticity, confidentiality, continuity, and freshness of checkpointed data. 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 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. We demonstrate a prototype implementation of SICP on an MSP430 microcontroller, and we investigate the overhead of SICP for several cryptographic kernels. To the best of our knowledge, this is the first work to provide a robust solution to secure intermittent computing.**

***Keywords*-intermittent computing, secure checkpoints, embedded systems, atomicity, continuity**

I. Introduction

Conventional embedded systems respond to power loss by losing its volatile state variables and rebooting from the initial state when power is restored. Embedded devices powered by energy harvesters, where power loss is a fact of life, utilize intermittent computing techniques to ensure forward progress of long-running applications [1–5]. Such systems create *checkpoints*, snapshots of the volatile program state stored in non-volatile memory, which includes the CPU registers, stack, peripheral states and all the application variables that are necessary to restore the program state. When the power is restored, the program state is reconstructed from the checkpoint and the program continues execution. If we treat power loss as an adversarial event, unprotected checkpoints pose a significant risk.

Risks of Checkpoints: When the checkpoint data may be accessed by the adversary, for example, through the microcontroller debug interface, the adversary can exploit the checkpoints in several ways, as listed in Figure 1. First, the adversary can read the checkpoint data, which reveals sensitive information such as the internal state of a cryptographic algorithm, leading to loss of confidentiality. Second, the adversary can tamper the checkpoint and take control over the embedded system after power is restored. Third, when the *same* checkpoint can be restored infinitely, the adversary can force repeated execution of the same section of code, as illustrated in Figure 1. A system moves through a sequence of activities symbolized through ON-states, during which a checkpoint is generated and stored in non-volatile memory. During a power transition after ON_2, if an adversary records and restores `CKP1`, the section of code in ON_2 is repeated.

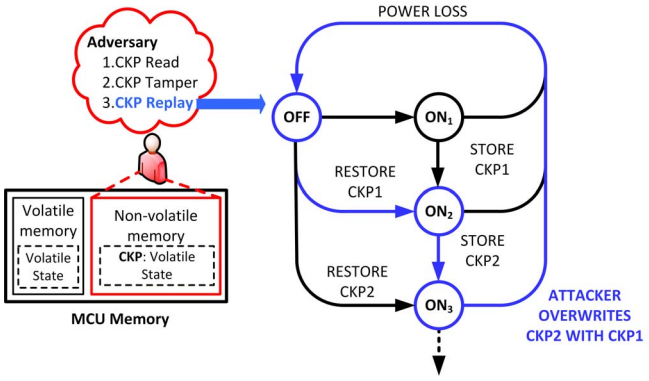

Figure 1: Contents of microcontroller (MCU) memory including the checkpoint, `CKP`, of an intermittent system which is vulnerable to an adversary and state diagram of such a system under replay attack. An adversary is able to repeatedly overwrite the new `CKP2` with the stale `CKP1` forcing continued re-execution of the code between ON_2 and ON_3.

Objectives: We identify four fundamental objectives, which will be achieved by SICP, to address these risks. For an intermittent system, the first three objectives pertain to its security and the last objective pertains to its availability.

1) *Information Security*: Information security ensures the confidentiality, integrity, and authenticity of checkpointed data.
2) *Freshness*: Freshness assures the system that the checkpoint to be restored is the most recent checkpoint and not a replayed checkpoint.
3) *Continuity*: Application continuity is the assurance that an application will resume execution from where it left off after a power loss without modification.
4) *Atomicity*: Atomicity guarantees that the protocol operations do not leave the intermittent system with an

Type of solution	Related Work	Essential properties						Target Platform
		C	I	A	Freshness	Continuity	Atomicity	
Intermittent computing	[1–4]	-	-	-	-	-	-	Embedded device
	Ghodsi [5]	✓	-	-	-	-	-	
NVM memory encryption	iNVM [6], SPE [7]	✓	-	-	-	-	-	Conventional computer
State continuity	ICE [8]	✓	✓	-	✓	✓	-	Conventional computer with protected module
	Ariadne [9]	✓	✓	-	✓	✓	✓	
	Memoir [10]	✓	✓	-	✓	✓	✓	
Secure checkpoints	SECCS [11]	✓	✓	✓	-	-	-	Embedded devices
	SICP(this work)	✓	✓	✓	✓	✓	✓	

C: Confidentiality, I: Integrity, A: Authenticity.

Table I: Comparison of the essential properties of SICP with related work

invalid state in the event of power loss during protocol execution.

Related Work: Table I 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 [1–4], except one [5], which only considers confidentiality and does not detect checkpoint replay. In-band memory encryption techniques have been proposed for conventional computers [6, 7], 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 [8], Ariadne [9] and Memoir [10] 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 [11]. It does not consider replay of checkpoints or availability of the intermittent device. As a result, SECCS does not ensure freshness of checkpoints and is not an atomic solution.

Contributions: We propose the Secure Intermittent Computing Protocol, referenced as the SIC Protocol or SICP. It guarantees all the essential properties listed in Table I by the following considerations. First, SICP incorporates freshness to checkpoints in the form of a nonce to detect replay of checkpoints illustrated in Figure 1. Second, SICP protects the information security of checkpoints using a device unique key. Third, all checkpoints are cryptographically linked to preserve the application continuity across power loss. Continuity guarantees the order of the sequence of checkpoints and preserves the run-time security properties in intermittent systems. Finally, SICP uses multiple stored states to guarantee atomic checkpoint generation and restoration, making SICP itself resilient to power loss. Unlike the other state continuity solutions listed in Table I, this protocol is designed for embedded systems. It is a bare metal solution that can be utilized by any embedded device with non-volatile memory. It does not require dedicated secure hardware support such as protected modules to guarantee the essential properties. We believe our solution is the first to provide comprehensive security to the checkpoints of an embedded device.

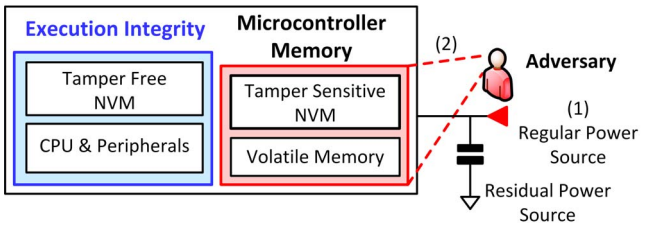

Figure 2: The architectural assumptions and memory model for SICP illustrating the assumed attacker model with two capabilities - (1) control power supply to the device and (2) view and modify tamnper sensitive non-volatile memory during power-off periods.

Organization: Section II explains the design and provides a brief overview of SICP, while Section III describes the protocol in detail. Section IV demonstrates the feasibility of the SIC Protocol through a prototype implementation on an MSP430 microcontroller, followed by our results in Section V. We conclude with Section VI.

II. Design

Threat Model: We assume an I/O attacker model, with two capabilities, illustrated in Figure 2. First, the adversary has complete control of the power supplied to the device. This gives the adversary the ability to arbitrarily stop the execution of the target program. Second, the adversary has access to the majority of the device memory during power-off periods except for a small portion of non-volatile memory, which is tamper free. The adversary can view and modify the device memory to read, tamper, or replay checkpoints except within the tamper free region. We assume a protected embedded software execution environment which provides execution integrity and memory protection when the device is powered on. The feasibility of this assumption has been demonstrated by recent efforts in attestation and isolation for microcontrollers [12]. We do not deal with the

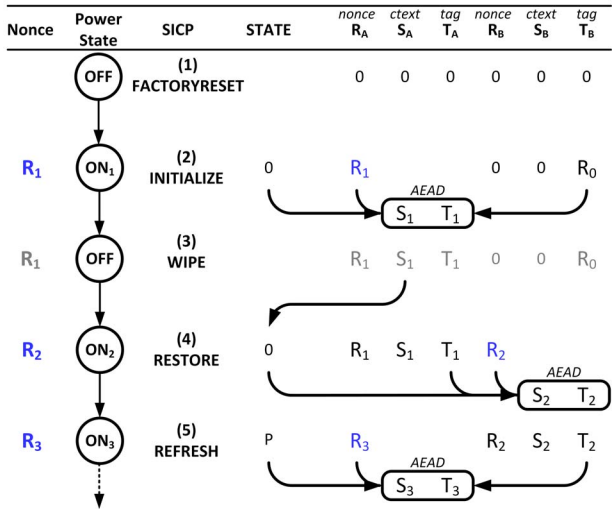

Figure 3: Example of the SIC Protocol. (1) The system is cleared by the *factory_reset()* operation. A fresh nonce, R_i is associated with each power-on state. (2) The first valid state save packet, SS_1, is created by the `INITIALIZE`. On power loss, (3) `WIPE` clears the volatile *STATE* and upon subsequent 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.

mitigation of side channel and fault injection attacks which are beyond the scope of this work.

Architectural Assumptions: Based on the above threat model, SICP requires certain basic capabilities from the microcontroller architecture, illustrated in Figure 2. The SICP architecture contains three types of memory. Volatile memory holds the volatile program state and is erased upon power loss. Tamper-sensitive non-volatile memory stores the secure checkpoints created from run-time program state. This non-volatile memory does not possess any tamper-resistance and represents the vast majority of the system's non-volatile memory. SICP also requires a small tamper-free non-volatile memory to store SICP variables that need tamper-free storage. At a 128-bit security level, SICP utilizes 48 bytes of tamper-free storage for two 128-bit nonces and a 128-bit secret key. The size of tamper free memory must be minimized to reduce hardware cost and complexity. Finally, we assume that the microcontroller has a residual power source which provides a small, finite energy supply when the regular power source is interrupted. This residual power source can, for example, be provided through power conditioning capacitors. We assume that the residual power source can be physically protected and can provide the minimum required energy to finish writing a 128-bit value to non-volatile memory and wipe sensitive program state.

A. SICP Operation

Figure 3 illustrates a working scenario of SICP. An intermittent system moves through a series of power transitions, represented by ON-states and OFF-states. The freshness objective is satisfied by assigning a nonce, R_i, to each power-on state of the device. The nonce is stored in tamper-free non-volatile memory to prevent checkpoint replay. It introduces freshness to the state, even if the application state is identical to a previous power cycle. For example, in Figure 3, a new nonce R_2 is associated with ON_2 even though the device is in the same application state as in ON_1.

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: $T_A \leftarrow AEAD_{auth}(S_A, T_B, R_A, K)$

Information security of the checkpointed data is protected using Authenticated-Encryption with Associated-Data (AEAD) [13]. The encryption function takes the plain text checkpoint, $STATE$, a nonce, R_i, and non-confidential associated data, T_i as input to generate the resultant encrypted checkpoint, S_i, and an authentication tag, T_i. S_i and T_i are stored in the device's tamper sensitive non-volatile memory. Similarly, the encrypted checkpoint is decrypted and restored onto $STATE$ when power is restored. With this structure, a secure checkpoint contains a tuple of R_i, S_i, and T_i, which we call a state save packet, SS_i.

The atomicity objective is satisfied using two state save packets, SS_A and SS_B. They are updated in an alternating manner, as shown in Figure 3, to keep one packet valid at all times. SICP is made resilient to power loss by atomic generation and restoration of checkpoints.

The continuity objective is satisfied by using the authentication tag from the previous state save packet as the associated data in the AEAD operations to generate the next state save packet. For example, in Figure 3, T_1 is used as associated data to compute T_2, which in turn is used as associated data to compute T_3. This process, referred as *tag-chaining*, cryptographically chains all the checkpoints together in a chronological order. Thus, authentication tags protect the authenticity and integrity of checkpoints as well as the order of the checkpoints.

Functions: SICP uses the following functions in its implementation. *factory_reset()* restores the device to manufacturer settings and loads a device unique key. $nonce()$ generates a unique and fresh nonce. $abort()$ flags a violation of SICP. $AEAD_{encr}()$ encrypts a checkpoint and $AEAD_{decr}()$ decrypts it. $AEAD_{auth}()$ generates the authentication tag. Typically, the AEAD interface for encryption/decryption returns both the ciphertext/plaintext and authentication tag. They are separated here into $AEAD_{encr}()$, $AEAD_{decr}()$ and $AEAD_{auth}()$ to provide clarity in protocol description.

III. SIC Protocol

SICP is defined as a collection of four algorithms: `INITIALIZE`, `REFRESH`, `RESTORE` and `WIPE`, defined as follows.

Algorithm 2 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**

INITIALIZE: This function is called the first time the device is powered on and is detailed in Algorithm 1. It is used to generate the first state save packet of the application, SS_A, following a *factory_reset()*. Since the first state save packet has no previous authentication tag to use for associated data, the tag, T_B, is initialized with a nonce before it is used to generate SS_A. This ensures a unique chain of tags after each *factory_reset()*. Finally, *STATE* is overwritten with zeros to ensure future tests for *factory_reset()* fail and INITIALIZE is only executed once after a *factory_reset()*.

REFRESH: Since the process of generating and restoring a checkpoint is similar, they are both described in one procedure, Algorithm 2. REFRESH can be called at any point by the application when the device is powered-on after INITIALIZE has finished generating the first valid state save packet. A state save packet is valid if it satisfies the following conditions. First, its nonce, R_i, must match the nonce used in the $AEAD_{encr}$ and $AEAD_{auth}$ operations. Second, the associated data used in these AEAD operations must match the tag of the previously valid state save packet. This ensures that at any point, only one state save packet is valid.

REFRESH will determine which is the latest state save packet, to update the alternating state save packet. For example, if SS_A was the latest packet to be refreshed, then when REFRESH is called again, line 2 in Algorithm 2 would be true. Correspondingly, the microcontroller will start updating SS_B by first updating R_B and then S_B. As long as T_B is not updated, SS_B does not yet contain a valid state save packet and SS_A remains valid. As soon as T_B is updated, it simultaneously invalidates SS_A and makes SS_B the latest valid state save packet. This write to T_B makes REFRESH atomic. The implementation of SICP makes an explicit assumption regarding the tag update in lines 8 and 16 of Algorithm 2. The tag update must be an atomic operation. The feasibility of this assumption is discussed in Section IV-A using the residual power source.

RESTORE: RESTORE is called upon every power-up, except immediately after a *factory_reset()*, restores the most recent valid *STATE* of the microcontroller. This function operates in the same manner as REFRESH with the difference listed on lines 4 and 12 of Algorithm 2. If there is a valid state save packet, the $AEAD_{decr}()$ function decrypts the ciphertext to restore it in *STATE*, otherwise, the program is aborted. SICP documents every power event in the sequence of checkpoints by generating a new state save packet upon every power-up. For example, if SS_B is valid, S_B is decrypted and restored in $STATE$, SS_A is updated with this *STATE*, a new nonce, Q and T_B. At the end of RESTORE, SS_A is made valid, invalidating SS_B. Thus, SICP ensures that no two different ON-states of the device are represented by the same element in the sequence of checkpoints.

If the stored state is tampered during power-off, the conditional checks on lines 2 and 10 fail, which flags a security exception and calls *abort()*. At a minimum, this function should either end program execution or clear the device memory and restart the device.

WIPE: WIPE must be called as soon as power loss is detected and the device is about to shut down. It clears transient information such as the program variables stored as plain text using the residual power source. The function must overwrite the sections of non-volatile memory that contain sensitive data, *STATE*. Volatile memory is wiped to prevent cold-boot style attacks [14]. The residual power source must have sufficient power to finish this operation otherwise the protocol will fail to protect the confidentiality of checkpoints. The specific implementation of WIPE will be device dependent. Section IV-A outlines our implementation's approach.

IV. Implementation

To evaluate the SIC Protocol, we establish an intermittent system and implement SICP to secure the system's checkpoints. We use a modified version of TI's *Compute Through Power Loss* (CTPL) utility [15] as the intermittent computing solution. The proof-of-concept implementation is created on an MSP430FR5994 Launchpad development board [16], which is equipped with 256kB of on-chip non-volatile ferroelectric RAM (FRAM). We use two different AEAD schemes. First, a software implementation of block cipher based EAX [17], provided by the Cifra [18] cryptographic library. Second, KETJE v2, specifically KETJE SR, a lightweight one-pass AEAD scheme, from the Keccak Code Package (KCP) [19].

A. Secure Intermittent Computing Support

Figure 4 illustrates the working of our implementation of SICP. The CTPL utility is modified to support user declared system checkpoints and to invoke the protocol functions

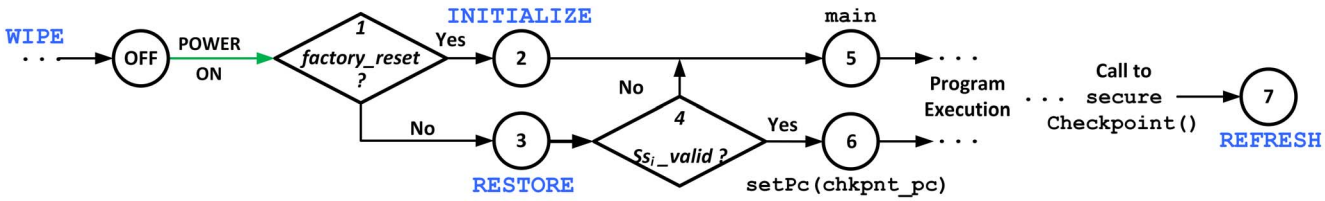

Figure 4: Our implementation of SICP. (1) SICP checks for *factory_reset()* and calls (2) `INITIALIZE` or (3) `RESTORE` to populate $STATE$ in non-volatile memory. `RESTORE`(4) inspects the non-volatile memory for a valid state save packet by checking the authentication tags, restoring it (6) if one 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 by calling `secureCheckpoint()` which in turns calls (7) `REFRESH` to generate a new checkpoint.

Component	Size (B)
Checkpoint Support	2532
EAX (HW)	3938
KETJE SR	3336

Table II: Executable Size Overhead

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

Table III: Energy and time overhead for SICP

within the checkpoint and startup process. `INITIALIZE` and `RESTORE` are called automatically during system startup, as shown in Figure 4. `WIPE` is also automatically triggered upon power loss. Only `RESTORE` is called during program execution by the user declared checkpoint function `secureCheckpoint()`. The application code calls `secureCheckpoint()` which in turn invokes `REFRESH` to generate a secure checkpoint.

Atomicity Support: The atomicity of the `secureCheckpoint()` function is ensured by using two state save packet buffers, SS_A and SS_B. All changes to non-volatile memory are made to the alternate buffer, such that the most recent state save packet remains valid until the newly computed tag is ready. 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 is achieved by disabling all interrupts for the copy duration of 48 cycles and relying on the residual energy of the device to ensure that even if power is lost, the copy operation will complete before the system stops operating. Thus, `secureCheckpoint()` completes the protocol operation without incident, ensuring the availability of the intermittent device.

nonce(): A majority of the nonces used in this protocol are provided by a 128-bit counter that is initialized to a random number during the `INITIALIZE` process and incremented each time a new nonce is requested. The exception is for the nonce for T_B in the `INITIALIZE` function. 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()*.

WIPE: The implementation of the `WIPE` operation requires detection of power loss by monitoring the device's V_{cc}. This is accomplished with MSP430FR5994's ADC12_B analog-to-digital converter, measuring V_{cc} against the system's V_{ref} as described in TI's FRAM Utilities [15]. 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 following the trigger [16]. When V_{cc} falls below V_{ref}, ADC12_B triggers the overwrite of *STATE* and SRAM via direct-memory-access using the residual power source.

V. 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 [16]. 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 II 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 III. SICP with 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 II and III.

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.

VI. CONCLUSION

This paper was motivated by the lack of appropriate security features incorporated in existing intermittent systems. The Secure Intermittent Computing Protocol addresses the security of intermittent systems across periods of power loss and its implementation highlights the heavy computational cost required to secure a stored system state. It is a generic protocol that can be used on top of any intermittent computing solution to secure its checkpoints. Our work demonstrated the need for future work in lightweight cryptographic kernels that can support AEAD schemes. In the future, a platform could be developed with a low latency non-volatile memory and the necessary hardware acceleration to support secure storage of larger system states.

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

REFERENCES

[1] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac, "Peripheral state persistence for transiently-powered systems," in *Global Internet of Things Summit, GIoTS 2017, Geneva, Switzerland, June 6-9, 2017*, 2017, pp. 1–6.

[2] H. Jayakumar, A. Raha, W. S. Lee, and V. Raghunathan, "QuickRecall: A HW/SW Approach for Computing across Power Cycles in Transiently Powered Computers," *JETC*, vol. 12, no. 1, pp. 8:1–8:19, 2015.

[3] M. Hicks, "Clank: Architectural Support for Intermittent Computation," in *Proc. of the 44th Annual Inter. Symposium on Computer Architecture, ISCA 2017*, 2017, pp. 228–240.

[4] 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 100vdd=0v with lt;400ns wakeup and sleep transitions," in *ISCC2013*, Feb 2013, pp. 432–433.

[5] Z. Ghodsi, S. Garg, and R. Karri, "Optimal checkpointing for secure intermittently-powered IoT devices," in *2017 IEEE/ACM Inter. Conf. on Computer-Aided Design (ICCAD)*, Nov 2017, pp. 376–383.

[6] S. Chhabra and Y. Solihin, "i-NVMM: A secure non-volatile main memory system with incremental encryption," in *38th Inter. Symp. on Computer Architecture (ISCA 2011*, 2011, pp. 177–188.

[7] S. Kannan, N. Karimi, O. Sinanoglu, and R. Karri, "Security Vulnerabilities of Emerging Nonvolatile Main Memories and Countermeasures," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and System*, vol. 34, no. 1, pp. 2–15, Jan 2015.

[8] R. Strackx, B. Jacobs, and F. Piessens, "ICE: A passive, high-speed, state-continuity scheme," in *Proc. of the 30th Annual Computer Security Applications Conf.*, ser. ACSAC '14. New York, NY, USA: ACM, 2014, pp. 106–115.

[9] R. Strackx and F. Piessens, "Ariadne: A minimal approach to state continuity," in *25th USENIX Secur. Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 875–892.

[10] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune, "Memoir: Practical state continuity for protected modules," in *2011 IEEE Symp. on Secur. and Priv.*, May 2011, pp. 379–394.

[11] E. Valea, M. D. Silva, G. D. Natale, M. Flottes, S. Dupuis, and B. Rouzeyre, "SECCS: SECure Context Saving for IoT devices," in *13th Inter. Conf. on Design & Techn. of Integrated Systems In Nanoscale Era, DTIS 2018*, 2018, pp. 1–2.

[12] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling, "Sancus 2.0: A Low-Cost Security Architecture for IoT Devices," *ACM Trans. Priv. Secur.*, vol. 20, no. 3, pp. 7:1–7:33, Jul. 2017.

[13] P. Rogaway, "Authenticated-encryption with associated-data," in *Proc. of the 9th ACM Conf. on Computer and Communication Security, CCS 2002*, 2002, pp. 98–107.

[14] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. 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*, 2008, pp. 45–60.

[15] *MSP MCU FRAM Utilities*, 2017. [Online]. Available: http://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/FRAM_Utilities/latest/exports/FRAM-Utilities-UsersGuide.pdf

[16] *MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family User's Guide*, 2012, no. SLAU678M. [Online]. Available: http://www.ti.com/lit/ug/slau367o/slau367o.pdf

[17] M. Bellare, P. Rogaway, and D. Wagner, *The EAX Mode of Operation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 389–407.

[18] J. Birr-Pixton, "Cifra: Cryptographic primitive collection," https://github.com/ctz/cifra, 2017.

[19] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer, "Keccak code package," 2017.