Discovering the Hidden Anomalies of Intermittent Computing

Andrea Maioli*, Luca Mottola*†, Muhammad Hamad Alizai+, Junaid Haroon Siddiqui+
*Politecnico di Milano (Italy), †RI.SE (Sweden), +LUMS (Pakistan)
Contact e-mail: andrea1.maioli@polimi.it

Abstract

Energy harvesting battery-less embedded devices compute *intermittently*, as energy is available. Intermittent executions may differ from continuous ones due to repeated executions of non-idempotent code. This anomaly is normally recognized as a "bug" and solutions exist to retain equivalence between intermittent and continuous executions. We argue that our current understanding of these "bugs" is limited. We address this issue by devising techniques to comprehensively identify where and how intermittent and continuous executions possibly differ and by implementing them in ScEpTIC: a code analysis tool for intermittent programs. Thereby, we find execution anomalies and their manifested impact on program behavior in ways previously not considered. This analysis is enabled by ScEpTIC design, implementation, and performance. ScEpTIC runs up to *ten orders of magnitude* faster than the baselines we consider, enabling many types of analyses that would be otherwise impractical.

1 Introduction

Energy harvesting is enabling a battery-less Internet of Things (IoT) of resource-constrained devices with small form factors [17, 34, 35, 39]. However, energy supply from the environment is generally erratic, causing frequent and unanticipated device shutdowns. For example, harvesting ambient RF energy for the execution of a simple CRC calculation leads to 16 power failures over a 6 seconds period [32, 6]. Executions thus become *intermittent*, as they consist of intervals of active computation interleaved by periods of recharging energy buffers.

Existing systems rely on small capacitors as energy buffers and on persistent state to ensure forward progress. Many solutions target mixed-volatile platforms, which facilitate handling persistent state as they map slices of the address space to non-volatile memory (NVM) [38, 20, 23].

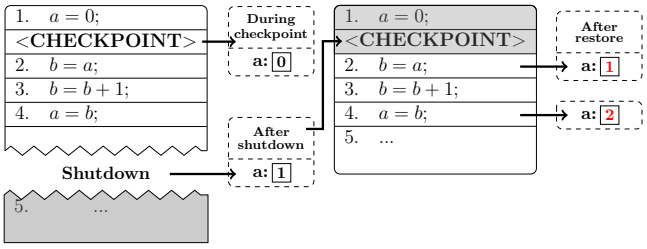


Fig. 1: Ther re-execution of line 2 incorrectly updates variable `a` allocated on NVM, leading to a memory anomaly.

Explicit *checkpoints* create persistent duplicates of volatile data, including registers and program counter.

Intermittent executions on mixed-volatile platforms introduce the possibility of *execution anomalies* [9, 31, 23, 38, 29], where programs reach states unattainable in a continuous execution. Anomalies may, for example, occur in *memory* due to hazardous read/write patterns caused by the re-execution of non-idempotent code. Fig. 1 shows an example. Variable `a` is allocated on NVM. A checkpoint occurs after line 1. Lines 2 to 4 eventually modify the value of `a`. The execution continues until power fails. When energy is back, the execution resumes with the state of volatile data from the checkpoint, that is, it restarts from line 2. However, `a` being on NVM, it retains its value from line 4 *before* the power failure, that is, the value produced by a later instruction compared to where execution resumes *after* the power failure [31]. Lines 2 to 4 increment `a` again, producing a different result than a continuous execution.

As we elaborate in Sec. 2, this type of memory anomaly is caused by a specific pattern of `load-store` memory accesses that creates a write-after-read (WAR) hazard. This anomaly is arguably the only one the literature distinctly acknowledges [38, 25, 23, 10, 24, 29]. Existing solutions remedy the problem with custom programming abstractions or compile-time techniques to retain equivalence between intermittent and continuous executions [38, 25, 23, 10, 24]. A few efforts also exist that aim to locate these anomalies and to provide guidelines to programmers for refactoring code [29].

We aim at gaining a deeper understanding of how intermittence affects program behavior. In Sec. 3, we describe techniques to exhaustively check the presence of memory anomalies. Using ScEpTIC, we demonstrate that intermittent programs are vulnerable to a wide variety of memory

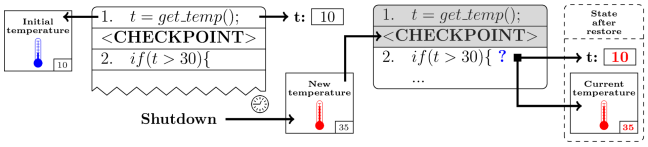


Fig. 2: Accessing a stale temperature reading.

anomalies, beyond those the literature commonly considers. The anomalies we recognize still originate from WAR hazards, and possibly manifest from a disparate set of memory access instructions, such as stack `push-pop` and function `call-ret`. Little discussion exists on these issues [29].

Execution anomalies due to *environment interactions* are also possible and generally harder to ascertain [8, 2, 5]. A power failure may cause programs to process stale environment state, such as an old sensor reading, or to perform unexpected actions on the environment, such as wrongly moving a rotor multiple times. Fig. 2 shows an example. Suppose a system suffers an unpredictably long power failure immediately after the execution of line 2. When the system resumes, the temperature might have changed, but the `if`-condition still evaluates to false with the old value of `t`. However, we can construct a different example where stale data may be valuable; for example, to compute long-term averages.

Determining whether and how execution anomalies affect environment interactions requires analyzing the causal impact of intermittence on the latter. The literature currently lacks the concepts and tools for this. In Sec. 4, we describe custom abstractions to qualify different types of environment interactions and implement proper support in ScEpTIC. Our tool tracks accesses to memory locations of interest and recognizes when a program is vulnerable to processing stale environment data. ScEpTIC also keeps track of the evolution of the environment state, for example, as determined by actuation, and determines if repeated executions of certain output actions can produce undesirable states.

Efficiently enabling the required code analysis is a challenge. A static analysis of the program would not provide run-time information required for analyzing the memory and environment. Checking actual executions in principle requires to analyze any possible combination of checkpoint placement and number of (re-executed) instructions. Running programs on target hardware is therefore plainly impractical, whereas source-level simulation may miss relevant read/write patterns that only manifest in machine code.

It would then appear that machine-level emulation is the only viable choice. That is, however, likely inefficient. A simple *CRC* computation [38] includes $5 \cdot 10^4$ machine-code instructions. If we were to test all possible combinations of checkpoint placement and number of (re-executed) instructions, we would need to analyze $2.34 \cdot 10^{13}$ machine-code instructions. As an example, our prototype emulator runs $5 \cdot 10^4$ instructions per second on a modern PC, which would mean 14 years for testing *CRC* computation.

Our tool ScEpTIC, described in Sec. 5, makes analysis of intermittent programs practical. ScEpTIC helps both system designers and developers analyze various programs, memory configurations, and forward progress mechanisms, to identify the most efficient configurations. System designers may also rely on ScEpTIC to evaluate different strategies for their forward progress mechanisms, which may be tuned accordingly to ScEpTIC results. Our tool is based on custom techniques we devise for analyzing memory anomalies as well as environment interactions. It takes LLVM intermediate-representation (IR) instructions as input to retain platform independence, and captures *all* occurrences of program anomalies due to intermittence, the conditions that cause them, and the effects they bear on program behavior.

In Sec. 6, we quantify the performance of ScEpTIC across different benchmarks and memory configurations. We compare ScEpTIC against a baseline that applies a *brute-force* approach to exhaustively analyze any possible intermittent execution as a function of checkpoint placement, interaction with the environment, and point of power failure. We show that ScEpTIC is up to *ten orders of magnitudes* faster. This means returning the results of code analysis in a matter of minutes rather than hundreds of days, enabling many types of investigations that would be otherwise impractical.

We end the paper in Sec. 7 with brief concluding remarks. ScEpTIC is available as open-source software [28].

2 Background and Related Work

We provide here the necessary background and an account of related work.

Mixed-volatile platforms. Low-power microcontroller units (MCUs) normally employ traditional SRAM as main memory. Thus, power failures cause a complete loss of state.

Frequent power failures motivate the design and manufacturing of mixed-volatile MCUs [30], where slices of the address space map to non-volatile memory facilities, such as FRAM. Data mapped to FRAM do not need to be checkpointed, as they are already persistent, thus sparing the corresponding overhead. This comes at the expense of increased energy consumption and slower memory access during normal operation [16]. FRAM-equipped MSP430 MCUs, for example, increase energy consumption by 2-3$\times$ compared to their volatile memory counterparts, and the MCU may only operate up to half of the maximum frequency without introducing waiting states to synchronize memory accesses [30].

Most importantly, registers and program counter, in addition to any volatile slice of main memory, need to be checkpointed anyways. The dichotomy between non-volatile and volatile memory spaces creates many of the issues we tackle.

Forward progress. Existing checkpoint systems focus on striking a trade-off between postponing the checkpoint; for example, to leverage new ambient energy, and anticipating it to ensure sufficient energy is available to complete it.

For example, Hibernus [3] and Hibernus++ [4] employ specialized hardware support to monitor the energy left. They operate in a *reactive* manner: whenever available energy falls below a threshold, they *react* by firing an interrupt that preempts the application and forces the system to take a checkpoint. Checkpoints may thus take place at *any* arbitrary point along the execution of a program.

Systems such as Mementos [32], HarvOS [7], and Chinchilla [25] employ compile-time strategies to insert specialized system calls to check the energy buffer. These triggers

bind checkpoint operations with a certain condition; for example, a checkpoint is only taken if available energy voltage falls below a threshold. Checkpoints thus happen *proactively* and *only* whenever the execution reaches one of these calls.

A similar duality exists in the solutions available to interact with the environment in intermittent programs, as two approaches exist. The *preventive* method seeks to achieve atomic interactions with the environment, and only initiates them when the remaining energy guarantees completion [21, 12]. Differently, the *recovery* method represents the evolution of peripheral states in main memory to bring the system back to a consistent state when resuming computations [8, 5, 2, 26]. Both of these methods integrate equally well with the techniques we explain next.

Debugging intermittent programs. Tools exist for the general problem of debugging intermittent programs, regardless of execution anomalies. For example, Ekho [15] recreates energy harvesting patterns to enable repeatable in-lab tests. CleanCut [11] identifies *non-termination bugs* in systems using task-based programming with transactional semantics.

Somehow closer to our work are EDB [9] and Siren [14]. In addition to traditional debugging features, EDB can emulate power failures and subsequent reboots. Siren introduces NVM and energy simulation capabilities in MSPSim. Using either tool, one may recognize a subset of the execution anomalies we identify by manually placing breakpoints and resets. This may be extremely laborious without apriori information, for example, a suspect of certain anomalies. Moreover, with Siren breakpoints and resets must be placed at the level of machine instructions and, unlike our work, neither tool provides any automated technique to cover all possible program executions that may manifest anomalies. They also do not consider environment interactions as we do.

Execution anomalies. Ransford and Lucia identify specific instances of memory-related execution anomalies [31]. Their insights provide a foundation for several later works that mask or avoid their occurrence [23, 10, 24, 38].

A specific analysis technique is presented by Van Der Woude et al. [38], who also solely acknowledge the same specific instances. They are, however, unable to assess the actual effects of anomalies and to recognize other instances, such as anomalies occurring on the heap.

Surbatovich et al. [37] identify a subset of the issues we identify for environment interactions. They assume that non-idempotent behaviors due to repeated I/O operations are to avoid, and thus provide a tool that determines the reach of input data through the program so developers fix these behaviors.

Our preliminary work on intermittence anomalies [29] covers only anomalies in main memory and on the stack. We extend our previous contribution with the support for anomalies happening on the heap. Different than our previous contribution, we also provide an analysis of environment interactions, which current literature overlooks, and a quantitative evaluation of our tool's performance.

3 Memory

We present techniques for *locating* memory anomalies and for evaluating their *effects* on program behavior. Both techniques are *sound* and *complete*, namely, they identify all and only the actual cases of memory anomalies.

3.1 Locating Anomalies

In general, memory anomalies due to intermittent executions may occur because of hazardous read/write patterns in NVM and depending on their interleaving with checkpoints.

To locate these anomalies, one should search for the conditions where a checkpoint occurs before a read on NVM, and there exist a write to the same NVM location before a following power failure. If so, a memory anomaly may occur due to WAR hazards, as in Fig. 1. This occurs because of the re-execution of read instructions after resuming, which may cause the program to load a value that was written by a later instruction, but before the power failure.

To be complete in identifying these cases, in principle, one should check all possible combinations of read/write operations on the same NVM address and all possible interleavings with checkpoint locations. For each different setting, one should execute the code for understanding how a given anomaly possibly propagates within the considered execution. As checkpoints might potentially occur at any point in the execution [3, 4], this creates an exponential increase in the number of possible executions that are to be checked, as discussed in the Introduction.

To address this issue, we determine the minimal amount of information necessary for the identification of memory anomalies and devise corresponding analysis techniques. These are based on the crucial observation that if one is only interested in *locating* these anomalies, looking for specific sequences of read/write accesses on NVM in a *single sequential* execution of the code suffices. Depending on the memory segment, these operations may take the form of `load/store`, `push/pop`, or `call/ret` pairs.

Note that we execute the program once to gather information that is usually not avaiable at compile time, such as the address of each accessed memory location, the evaluation of conditional instructions, and the executed branch paths. We then rely on developers to provide a sufficient set of tests that cover all execution paths that are input-dependent.

These techniques and their implementation in ScEpTIC eventually lead us to confirm current findings [31] and to recognize additional memory anomalies.

3.1.1 Data Access Anomaly

Fig. 1 is a case of *data access anomaly*, as reported in literature [31]. We recognize such an anomaly whenever x is a memory address in NVM and an ordered sequence of machine-code instructions $I_1, ..., I_n$ exists such that:

- I_1 loads a value from an address x,
- I_n modifies the value stored at address x,
- no checkpoint exists in the sequence $I_1, ..., I_n$.

These conditions entail that if a power failure occurs after I_n, the system resumes before I_1 which is then re-executed; I_1 then reads the value produced by I_n before the power failure, that is, from a later instruction. A fix for this is placing a checkpoint between I_1 and I_n to avoid re-executing the *load* operation when resuming [38].

Here, we reduce the information necessary for locating memory anomalies based on two key observations:

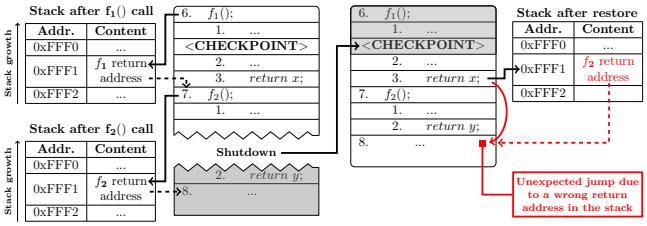


Fig. 3: Return address overwritten by call to f_2 showing an activation record anomaly.

1. any non-write access after I_1 need not to be checked separately, because the potential memory anomaly it may cause is already captured by the analysis from I_1;
2. only write accesses occurring in the sequence $I_1, ..., I_n$ meet the conditions to produce a WAR hazard; in fact, any other write access that follows a checkpoint after this sequence can not affect prior read accesses, and becomes part of a different sequence $I_{n+1}, ..., I_m$.

These two criteria form the basis to efficiently analyze other, previously unseen kinds of memory anomalies.

3.1.2 *Activation Record Anomaly*

We uncover executions whereby allocating the stack on NVM, upon resuming from a power failure, non-volatile information is read from the activation record of a function to be executed later. This *activation record anomaly* may lead to wrong results, unwanted jumps, or a program crash.

Fig. 3 shows an example. A call to function f_1 executes first and its activation record is placed on the stack. A checkpoint takes place after line 2 inside f_1. When f_1 returns, its activation record pops from the stack and execution continues from line 7. The stack content on NVM is not deleted when returning from f_1; only the stack pointer changes. When placing the activation record of f_2 on the stack, the one of f_1 is overwritten. If a shutdown happens during the execution of f_2, the execution resumes inside f_1 according to the checkpoint data, but the activation record is that of f_2.

Note that Fig. 3 shows the case where the return address from f_2 is read as the one of f_1 when execution resumes. This is only one of the possible outcomes. Worse is if f_2 overwrites f_1 return address with data representing an invalid address, such as a local variable or a saved register, causing a program crash when execution resumes. In general, the sequence of `pop` instruction belonging to the epilogue of f_1 may read the values produced by `push` instructions belonging to the prologue of f_2. Also, note that f_2 may equally be a programmer-defined interrupt handler that fires asynchronously, making the issue even more difficult to track.

We find that an *activation record anomaly* exists whenever the stack is allocated on NVM and an ordered sequence of machine-code instructions $I_1, ..., I_n$ exists such that:

- I_1 is a `call` instruction for function f_x,
- the execution of f_x includes at least one checkpoint,
- I_n is a `call` instruction,
- no checkpoint exists in the sequence $I_1, ..., I_n$.

The anomaly exists because a checkpoint is saved inside the context of a function f_1, f_1 returns, and a subsequent call

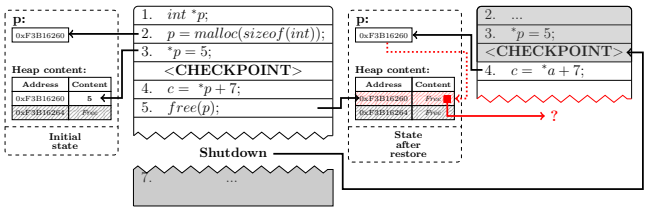


Fig. 4: Example of memory map anomaly.

to f_2 overwrites parts of the activation record of f_1. Checkpointing between the return of f_1 and the call to f_2 addresses the issue, preventing the execution from resuming inside f_1.

Here, we reduce the information to check for locating these anomalies by applying the two criteria in Sec. 3.1.1, but also noticing that the analysis need not to consider the code of f_x. Only the fact that f_x somewhere includes a checkpoint matters. We may analyze the code of f_x separately compared to the search of the conditions above, and no information from the analysis at the level of function calls need to percolate into the analysis of callees.

Ratchet [38] identifies a specific instance of the problem arising with interrupts. The general case is, however, overlooked in existing literature and may be recognized only by reasoning at the level of machine code, not source code.

3.1.3 *Memory Map Anomaly*

When read/write instructions on NVM involve operations that possibly change the heap state, a *memory map anomaly* occurs whereby a dynamic memory operation observes a future state of memory upon resuming from a power failure.

Fig. 4 shows an example. Line 2 allocates a heap block and saves its address in pointer p. A checkpoint occurs before line 5, which de-allocates the same memory block. If a shutdown happens after line 5, the execution resumes from line 4, whose memory access may now lead to unpredictable results [40] as the block was previously de-allocated.

It would be possible to construct arbitrary combinations of heap operations before and after a checkpoint, leading to this kind of anomaly. If pointer information are not updated, the re-execution targets the memory address before the shutdown, whereas the memory block may now be freed or re-allocated somewhere else.

We find that a *memory map anomaly* exists whenever the heap is allocated on NVM and an ordered sequence of machine-code instructions $I_1, ..., I_n$ exists such that:

- I_1 is a `load` or `store` instruction targeting the heap block pointed by x,
- I_n is a `free` or `realloc` instruction that modifies the heap block pointed by x,
- no checkpoint exists in the sequence $I_1, ..., I_n$.

The anomaly exists because pointer information are not consistent with the state of the heap. Properly placing checkpoints to avoid re-executing instructions based on possibly inconsistent pointer information solves the issue.

Similar to the stack, the two criteria in Sec. 3.1.1 are valid here too to help locate heap anomalies efficiently. In addition, we note how allocating the heap on NVM with a transactional memory controller [36] does not ensure atom-

icity for heap modifications, either. Power failures happening during the execution of any such instructions leave the heap state partially changed. The re-execution of instructions that perform destructive changes to the heap, such as `free` or `realloc`, is also a possible source of anomaly, whereas re-executing memory allocation operations, such as `malloc`, does not affect correctness but may yield memory leaks.

Existing literature overlooks the existence of this kind of anomaly too, which again may only be recognized by reasoning at the level of machine code and raw memory accesses.

3.2 Evaluating Effects

The observations above serve to recognize and locate memory anomalies, but they do not suffice to examine how their effects change the program state compared to a continuous execution. Information on this may be crucial for identifying the cause of a program crash or for performing a post-mortem analysis, as the change of behavior may, for example, corrupt the state in subtle ways and thus percolate throughout possible long-running executions [40].

To this end, a single sequential program execution can only provide partial information. We rather need to emulate the code re-execution, by pretending checkpoints at certain code locations are executed and power failures occur later. We crucially observe that we may use the conditions we identify in Sec. 3.1 also to reduce the number of locations where checkpoints and power failures need to be emulated. In essence, it is sufficient to first locate the anomaly and only then, to re-execute the relevant parts of code.

For example, consider analyzing data access anomalies according to the conditions in Sec. 3.1.1. To understand their effects, we create a new emulated execution starting at I_1 with the state that a continuous execution would have at that point, and proceed up to I_n where we pretend a power failure to happen. Then, we take the state of the NVM there, bring it back to I_1, and combine it with information in the checkpoint that we assume to occur right before I_1. We resume the execution as if the device had new energy and proceed again up to I_n. The program state at this point represents how the data access anomaly alters the program state. Similar techniques are applicable for all memory anomalies in Sec. 3.1.

4 Environment Interactions

Interactions with the environment are a key functionality of embedded sensing devices. As the notion of correctness here is application-specific, understanding how they affect intermittent executions requires to develop both appropriate abstractions and analysis techniques. The problem takes different forms for input (sensing) and output (actuation) interactions. When integrated with approaches to cope with power failures during the interaction itself, as explained in Sec. 2, the techniques we explain next apply to both methods using preventive and recovery techniques.

4.1 Input Interactions

Intermittent executions create a data-time dependency [18]. A piece of urgent data may expire after a long energy outage, requiring the system to sense again before resuming the execution. Old data may still be valuable depending on applications requirements; for example, in applications that are interested in long term trends.

Abstractions. We define two concepts to qualify how, according to the programmers' intentions, input environment data should be accessed in intermittent programs. Under a *most-recent* access model, a program is expected to access the input data only if it is gathered within the same power cycle, that is, no power failure occurs between the time the data is acquired and when it is used. This is the case where applications must take decisions based on the most up-to-date environment data. Differently, under a *long-term* access model, a program may access the input data independent of when it is originally gathered, that is, an arbitrary number of power failures may occur between when the data is acquired and when it is used. This is the case where data is valuable because of its long-term significance.

We ask programmers to tag individual variables storing sensor data as behaving according to either model. The techniques we illustrate next allow programmers to understand whether, depending on checkpoint placement, the semantics of their variables matches the required access model. Note that this analysis is meaningful for system support employing proactive techniques [32, 7, 25], as explained in Sec. 2. Programmers may move the placement of checkpoint calls to ensure that given variables behave according to the desired access model. Differently, in systems employing reactive techniques, checkpoints may happen anywhere in the code [3, 4]. Variables that store sensor data thus behave according to a *long-term* access model, because checkpoints might potentially happen anytime between when the data is gathered and when it is later used.

Analysis. We perform a single sequential execution of the code and use two additional bookkeeping data structures, a *checkpoint clock* and an *access record*. The former establishes an ordering of the events in the code and is incremented each time we pretend a checkpoint to happen. This corresponds to every location in the code where a call to the checkpoint routine is inserted. The access record tracks accesses to memory locations of interest.

We explain the process with the help of Fig. 2. Initially, the *checkpoint clock* is set to 0. After executing line 1, the *access record* for variable t is updated as $\langle t, temperature, 0\rangle$, where variable t contains the value of a given sensor when the checkpoint clock is 0. Next, the *checkpoint clock* is incremented by one as we encounter a further call to the checkpoint routine. Thereafter, the execution of line 2 leads to another update in the *access record* for t with $\langle t, temperature, 1\rangle$. The variable is thus accessed across checkpoint call, and thus behaves according to a *long-term* access model. If the latter differs from the access model the variable is tagged with, a warning is returned.

Note that memory buffers for sensed data on NVM may also suffer from memory anomalies, which can be tested with the techniques described in Sec. 3.

4.2 Output Interactions

Intermittence may cause an application to perform a stale, duplicate, or falsified action on the environment. Fig. 5 shows an example. Line 1 rotates the servo *relatively* by 45°. A power failure occurs right after line 1. When execution resumes from the checkpoint, the servo is rotated again by a

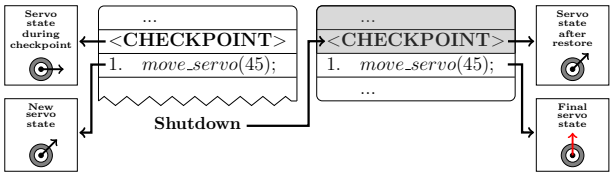


Fig. 5: The re-execution of instruction 1 yields an unexpected environment state.

further 45°, taking its current position to 90°. The outcome does not correspond to a continuous execution.

Abstractions. Similar to Sec. 4.1, we define two semantics for actuation commands: *absolute* or *relative*. The former models the cases of idempotent actuation commands. The latter models the opposite, that is, the resulting state of the environment is a function of the initial state and of actuation.

Application requirements dictate whether to rely on either semantics. Unlike Fig. 5, some applications may want to affect the state of environment whenever an actuation command is executed, regardless of the number of repetitions. Consider, for example, an application that sends an announcement whenever it wakes up from a power failure.

To enable code analysis, programmers are required to express the semantics they expect by tagging the individual calls to actuation commands as behaving according to either model. In addition, they are to provide an abstract specification of how the environment changes in response to the (possibly partial) execution of actuation commands. This specification is primarily meant to check that two environment states are semantically equivalent. This information is, in general, application-dependent. Vast literature exists on the subject [13]. We thus omit the description of such specification for brevity, which is nonetheless available [27].

Analysis. To understand whether intermittent executions match the expected actuation semantics, we execute the program until encountering an actuation command. There, we record the state of environment up to the point of the power failure, either during or after the command execution, according to the abstract environment specification. We re-execute the code from the previous checkpoint up to the same actuation command. We can now compare the new environment state with the previously recorded one.

If the states differ, the command behaves as *relative*, otherwise, it behaves as *absolute*. If this behavior does not match the programmers' expectations, a warning is returned. Programmers may now change the implementation according to application requirements. For example, they may replace an actuation command exposing a *relative* semantics with one implementing an *absolute* one, or build a wrapper around the former to achieve the desired behavior.

5 Implementation

We implement the techniques in Sec. 3 and Sec. 4 in a tool called ScEpTIC, which works in four different modes:

1. ScEpTIC-Locate performs the analysis to *locate* memory anomalies, as in Sec. 3.1.
2. ScEpTIC-Evaluate performs the analysis to *locate* memory anomalies and to evaluate *their effects*, as in Sec. 3.2.

Table 1: Consumers and producers of memory anomalies.

	Data Access - Sec. 3.1.1	Activation Record - Sec. 3.1.2	Memory Map - Sec. 3.1.3
Consumer	`load`	`ret/pop`	`load, store, realloc, free`
Producer	`store`	`call/push`	`malloc, realloc, free`

3. Environment-Inputs verifies the coherence of *input interactions* with programmer-specified semantics, as discussed in Sec. 4.1.
4. Environment-Outputs behaves symmetrically w.r.t. the previous option for *output interactions*, as discussed in Sec. 4.2.

We describe next the architecture of ScEpTIC and the details of the first two modes. The processing required for the other two options is a minimal variation of the former.

5.1 Architecture

ScEpTIC is written in Python and processes LLVM intermediate representation (IR) code to gain independence from specific platforms. It comprises two main modules: the *abstract-syntax tree (AST) builder* and the *emulator*.

The regular LLVM AST builder is augmented with architecture-specific components such as registers, libraries, and proxies for emulating environment interactions, as described next. The resulting AST is then translated to be executed by the emulator module.

ScEpTIC also allows users to annotate functions used for environment interactions. The annotation takes as input: *i)* the type of interaction, namely, input or output; *ii)* the name of the function that interacts with the environment; *iii)* a list of LLVM IR types, representing the function argument types; and *iv)* the type of return value and a logic to generate such values, for example, a generator function or a statically-defined list of values.

The ScEpTIC emulator models user-specified general registers and special purpose registers that exist on all platforms, such as the program counter (PC) and the stack base pointer (EBP). The emulator divides the available memory into three segments: the global symbol table (GST), the stack, and the heap. The GST segment is further subdivided into volatile and non-volatile regions, placing the global variables according to programmer's requirements.

5.2 Locating Memory Anomalies

We call *producer* (*consumer*) any instruction that alters (accesses) the content of NVM. By generalizing the concepts of Sec. 3, we argue that to locate a memory anomaly, we need to identify an ordered sequence of instructions $I_1, ..., I_n$, such that I_1 is a consumer, I_n is a producer, I_1 and I_n operate on the same NVM location, and no checkpoint occurs between I_1 and I_n. The pair $\langle I_1, I_n \rangle$ is the one causing the memory anomaly. Tab. 1 indicates the consumers and producers for the memory anomalies discussed in Sec. 3.

The processing we devise for locating memory anomalies only requires sequentially executing the program and collecting a trace of NVM memory states. To create the trace, when a producer or consumer is encountered, we save the state of the target memory locations, the value of an instruction counter that corresponds to its execution, the operation type among those of Tab. 1, and the program counter. We organize this information into a two-level dictionary that has the memory address as first key and the operation type as the

Procedure 1: Locating memory anomalies for a given NVM location.

```
1  function ScEpTIC-Locate (trace, consumer, producer, ED)
2      anomalies ← ∅
3      consumers ← trace[consumer]
4      producers ← trace[producer]
5      foreach pair ⟨counter, consumer_pc⟩ ∈ consumers do
6          window ← SlideWindow(producers, counter, ED)
7          foreach ⟨producer_counter, producer_pc⟩ ∈ window do
8              insert ⟨consumer_pc, producer_pc⟩ into anomalies
9      return anomalies

10 function SlideWindow (producers, consumer_counter, ED)
11     min_counter ← consumer_counter
12     max_counter ← consumer_counter + ED
13     return ∀ producer ∈ producers s.t. min_counter ≤
           Istruction_counter(producer) ≤ max_counter
```

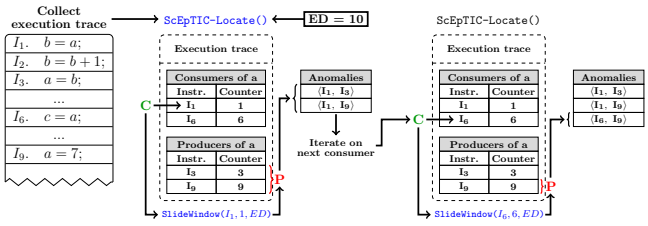


Fig. 6: ScEpTIC-Locate identifies memory anomalies in the execution trace by considering a sliding window of producer instructions altering a given variable.

second one. This allows for an efficient search of memory anomalies when traversing the trace.

Procedure 1 shows the core logic to process the execution trace for a given pair of producer and consumer that possibly cause a memory anomaly. Fig. 6 helps understand the processing with a concrete example. Starting from every consumer, that is, a candidate I_1 that accesses a given memory location (line 5), the procedure operates on a window of producer instructions determined by `SlideWindow` (line 6), with a corresponding instruction counter higher than that of the consumer, and as a function of the checkpoint strategy. In the example of Fig. 6, this extends up to I_9 for consumer I_1. We start from I_1 as we emulate a checkpoint immediately before it; in this case, every producer in the window is a potential I_n that causes a memory anomaly (lines 7-8).

The key for correct and efficient analysis rests in the interplay between `ScEpTIC-Locate` and `SlideWindow`. For the latter, Procedure 1 shows the case of reactive checkpoints, which can potentially happen anywhere in the code. Given the instruction counter corresponding to a consumer operation we consider as the first instruction after a checkpoint, the window extends for a number of instructions whose energy cost equals the energy left after the checkpoint operation and eventually leading to a power failure. These are the instructions that would be possibly re-executed upon resuming. We call this quantity *execution depth* (*ED*)

Actual meaningful values for *ED* depends on the device energy consumption, capacitor size, and checkpoint energy consumption. In Sec. 6.1 we show how to accurately calculate *ED*, as this is essential for obtaining accurate information on intermittence anomalies. Underestimating *ED* may cause the analysis not to identify some intermittence anomaly, whereas overestimating *ED* may cause the analysis to identify bogus intermittence anomalies.

As the analysis of the current window completes, `ScEpTIC-Locate` slides the trace down to the next consumer (line 5), which `SlideWindow` now considers the first instruction executing after a potential checkpoint, that is, a new candidate I_1. In Fig. 6, this happens to be the assignment `c=a`. Note how sliding the instruction window down to any instruction between the former I_1 and the new one does *not* uncover memory anomalies this procedure would not uncover, and thus represents unnecessary overhead. For example, the instructions between `b=a` and `c=a` in Fig. 6 are covered already by the first iteration of the procedure.

The case of proactive checkpoints is a simplified version of Procedure 1. Instead of considering every consumer as the first instruction executed after a potential checkpoint, we simply consider as the candidate I_1 the set of consumer instructions between every statically-inlined checkpoint call and the next producer I_n. Considering consumers I_{n+k}, $k > 0$ past the producer I_n is unnecessary, as they necessarily read the value produced by I_n, as in a continuous execution.

Accordingly, `SlideWindow` now stretches the window of producer instructions from I_1 up to the next statically-inlined checkpoint call, regardless of *ED*. This is the most conservative choice, as it assumes the system has just enough energy to execute every following instruction, but fails to complete the next checkpoint call. The number of possibly re-executed instructions is thus highest. When sliding the window down, `ScEpTIC-Locate` proceeds to the set of consumer instruction after the next checkpoint call, and the procedure repeats.

5.3 Evaluating Effects

The analysis of how memory anomalies possibly impact the program behavior requires the concrete emulation of power failures, with the corresponding code re-execution.

Consider again the case of reactive checkpoints; the case of proactive checkpoints is obtained as a variation of this, similar to Procedure 1. For the analysis to be complete, based on the observations illustrated earlier for locating memory anomalies, it suffices to investigate the case when checkpoints happen before every *consumer* instruction. This is a candidate I_1. The window of instructions that we re-execute extends for *ED* instructions starting from I_1. The instructions that possibly cause the memory anomaly are the producers I_n within this window, whereas the effects of the memory anomaly are manifest, for example, as a consumer instruction I_1 accesses an altered value when it is re-executed upon resuming after a power failure.

Procedure 2 shows the core logic for evaluating the effects of memory anomalies. The example of Fig. 7 helps understand the processing. Before emulating a consumer instruction operating on NVM, we run procedure `ScEpTIC-Evaluate`. It starts off by saving a snapshot of the emulation state, including the current instruction counter (line 2). This information is necessary to roll back the emulated execution to a consistent state in case no memory anomalies are found by re-executing the code from the considered

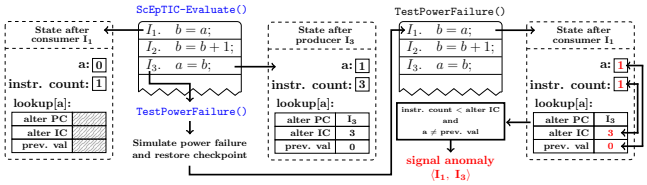


Fig. 7: ScEpTIC-Evaluate procedure to test a checkpoint before instruction I_1.

Procedure 2: Evaluating the effects of memory anomalies for a given NVM location.

```
1  function ScEpTIC-Evaluate (state, checkpoint_data, ED)
2      snapshot ← snapshot of state
3      target_counter ← InstructionCounter(state) + ED
4      lookup ← {}
5      while InstructionCounter(state) < target_counter do
6          pc ← ProgramCounter(state)
7          addr ← TargetAddress(pc)
8          old_content ← NVMContent(state, addr)
9          execute pc
10         if pc is consumer then
11             init lookup[addr] with empty values
12         if pc is producer and addr ∈ keys(lookup) then
13             current_counter ← InstructionCounter(state)
14             lookup[addr] ← ⟨current_counter, old_content, pc⟩
15             if no failure previously simulated after pc then
16                 ⟨state, lookup⟩ ←
                     TestPowerFailure(state, snapshot,
                     checkpoint_data, current_counter, lookup)
17 function TestPowerFailure (state, snapshot, checkpoint_data,
       target_counter, lookup)
18     simulate power failure
19     restore checkpoint_data
20     while InstructionCounter(state) < target_counter do
21         pc ← ProgramCounter(state)
22         execute pc
23         if pc is consumer then
24             addr ← TargetAddress (pc)
25             if addr ∈ keys (lookup) then
26                 val ← NVMContent (state, addr)
27                 ⟨counter2, value2, pc2⟩ ← lookup[address]
28                 if counter2 >
                       InstructionCounter(state) and val2 ≠ val
                       then
29                     signal memory anomaly ⟨pc, pc2⟩
30     if at least one anomaly was found then
31         lookup ← ∅
32         restore snapshot
33     return ⟨state, lookup⟩
```

consumer. Then, it calculates the length of the instruction window to analyze (line 3) and initializes the *lookup* information used for tracking the NVM state (line 4).

For every consumer operation, ScEpTIC-Evaluate initializes the *lookup* information associated to the target addresses (line 6-11). In Fig. 7, this is shown on the leftmost box for variable a. As the execution continues and a producer is found, ScEpTIC-Evaluate verifies if any *lookup* information is present for the target address (line 12). This may entail that an earlier consumer instruction can access an altered information in case of a power failure and subsequent re-execution. If so, we update the *lookup* information of the altered memory location (line 13-14). This is the case for producer I_3 in Fig. 7, as shown in the middle box.

If it is the first time we analyze a specific consumer/producer pair (line 15), we test the effects of a power failure at this point with TestPowerFailure. This resets the volatile state (line 18) and restores the checkpoint (line 19) with the instruction counter at the time of checkpoint. This effectively rolls back the execution to the consumer that triggered the processing, that is, I_1 in Fig. 7. It re-executes the code until it reaches the point of the earlier power failure. Whenever a *consumer* is executed (line 23), TestPowerFailure accesses the *lookup* information to verify if it accesses the value of a producer in the previous power cycle (line 25-29). TestPowerFailure thus identifies a memory anomaly. Fig. 7 shows this happening as soon as I_1 is re-executed, based on the information in the rightmost box.

By continuing the execution, TestPowerFailure assesses the effects that the memory anomaly causes on program behavior, including also other memory anomalies, and up to *ED* instructions from I_1. Upon completion, if any memory anomaly is found, TestPowerFailure restores the snapshot and empties the *lookup* information (line 30-32) before returning control to ScEpTIC-Evaluate. This allows the latter to proceed with the analysis from a clean consistent state, not altered by the effects of memory anomalies.

6 Evaluation

We evaluate our techniques using ScEpTIC on a system with an Intel Xeon E3-1270, 64 Gb of RAM, Ubuntu 19.04, and Python 3.7.2. We use Clang 5.0.1-4 with LLVM 5.0 [22] to produce the LLVM IR [22].

6.1 Memory Anomalies: Setup

We evaluate the performance of the techniques in Sec. 3 by comparing them to a baseline that operates only based on the conditions that possibly lead to a memory anomaly. In contrast, existing forward progress mechanisms [23, 24, 25, 38] avoid the occurrence of intermittence anomalies with analysis techniques that are strictly tied to their system or memory configuration. These may fail to identify the occurrence of anomalies with different memory configurations. As they operate at compile time, they also cannot identify the occurrence of intermittence anomalies that happen across branches, conditional operations, and dynamic memory accesses. Because of this, we use as baseline a "layman" approach that does identify the occurrence of *all* anomalies.

Baseline. Initially, Layman-Memory executes the code sequentially. Every time it needs to analyze a potential checkpoint location, it saves snapshot of the emulation state and then proceeds with the execution. Following the checkpoint location in the code, Layman-Memory records a snapshot of the memory state as the execution unfolds, until it emulates a subsequent power failure. Then, it rolls the execution back to the checkpoint location, emulates a resume operation, and proceeds with the (re-)execution by comparing the snapshots of the memory states produced by the (re-)executed code against those collected earlier.

An anomaly is found whenever a mismatch is detected, as the intermittent execution would be different from the continuous one. Because the comparison occurs on snapshots of the entire memory state, LAYMAN-MEMORY can only provide coarse-grained memory information and cannot pinpoint what instructions are responsible for a given anomaly.

Benchmarks and configurations. We select three benchmarks commonly used in intermittent computing [3, 32, 4, 7, 1]: Cyclic Redundancy Check (CRC) for data integrity, Fast Fourier Transform (FFT) for signal analysis, and Advanced Encryption Standard (AES) for security. They span diverse functionality and expose very different program structures. We use their open-source implementations from MiBench2 [19], that is, a benchmark suite already used for evaluating system support for intermittent computing [38].

For each benchmark, we choose two different memory configurations. One configuration places only *global variables* onto NVM, while allocating all other memory segments, including the stack, on volatile main memory; the other configuration places only the content of the *stack* onto NVM, while allocating all other memory segments, including global data, on volatile main memory.

We consider two different use cases. In the *a-priori* scenario, we use SCEPTIC at a time when the checkpoint strategy is yet to be defined, that is, programmers are to select the most suitable system support. This means checkpoint locations in the code are not known, or reactive checkpoint systems are employed that may preempt the execution at any point in time. For the analysis to be complete in this scenario, every possible checkpoint location should be examined along with any potential location of power failure.

In the *a-posteriori* scenario, we use SCEPTIC when the checkpoint strategy is fixed and checkpoint calls are statically placed in the code. This covers the cases where programmers aim to analyze a specific checkpoint placement [7, 32] or perform a post-mortem analysis of an already-deployed program. We consider the checkpoint placement of Mementos [32]. For AES, we consider both Mementos' *function-return* strategy that positions a checkpoint after the return of each function and its *loop-latch* strategy that places a checkpoint at the end of every loop body. We consider only the latter strategy for the CRC and FFT benchmarks, since they include no significant function calls.

Metrics. The primary performance figure we consider is the net *execution time* required for the analysis, as it determines how practical is a given technique. Moreover, to perform the analysis, a certain technique may need to emulate power failures and possibly re-execute certain instructions, resulting in an increase of the *number of embedded code instructions executed*. We measure this figure as well, as it impacts the execution time. Similarly, the different techniques also collect information about the program state into data structures that are external to the emulated program, so to verify the presence of anomalies. These *accesses to support memory* introduce an overhead that also influences the execution time, and is thus worth measuring as well.

Finally, we compare the number of found memory anomalies by the different systems we test as an indication of the *output noise*. As explained in Sec. 3, SCEPTIC-LOCATE is both sound and complete, and it also returns every memory anomaly as a unique data point, essentially providing the cleanest non-redundant output. Differently, SCEPTIC-EVALUATE may return a higher number of found memory anomalies merely because the effects of the same WAR hazard may differ at run-time based on actual data. We refrain to measure this metric for SCEPTIC-EVALUATE. On the other hand, LAYMAN-MEMORY may point to the same memory anomaly in multiple seemingly different ways, because coarse-grained memory information and the inability to identify the exact instructions responsible for a given anomaly prevents it from filtering out redundant information.

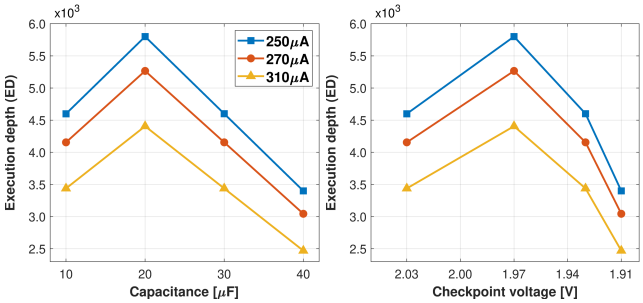


Fig. 8: Execution depth in Hibernus++ [4] with respect to current draw, capacitance, and checkpoint voltage threshold.

Code re-execution. In the *a-priori* use case where checkpoint calls are not statically placed in the code, we need to specify a realistic value for the ED parameter, discussed in Sec. 5.2, representing the number of instructions executed after a checkpoint and before the subsequent power failure.

We consider a configuration similar to the one of Hibernus++ [4], using MSP430 [30] MCUs. When the capacitor voltage goes below a certain threshold V_{trig}, Hibernus++ saves a checkpoint. Here, ED corresponds to the number of instructions that the MCU executes with the remaining energy, the checkpoint is complete and assuming the ambient provides no additional energy in the meantime.

The capacitor equipping most intermittently-computing systems is an electric bipole characterized by the differential relation

$$i(t) = C\frac{dv(t)}{dt}, \tag{1}$$

where C is the capacitance, $i(t)$ is the current at time t, and $v(t)$ is the capacitor voltage at time t. With a constant current draw, we state

$$\Delta t = C\frac{(V_2 - V_1)}{I} \tag{2}$$

to be the time required to discharge the capacitor from voltage level V_2 to V_1, with a constant current draw I.

Let us consider V_2 to be the voltage level V_{trig} where Hibernus++ [4] triggers a checkpoint and V_1 to be the voltage level V_{min} where the MSP430 powers off. We can express Δt as $t_{chk} + t_{cmp}$, where t_{chk} is the time required for saving a checkpoint and t_{cmp} is the remaining running time of the MCU. From eq (2) we derive

$$t_{cmp} = \Delta t - t_{chk} = C\frac{(V_{trig} - V_{min})}{I} - t_{chk}. \quad (3)$$

Given t_{cmp} as the remaining running time of the MCU, the number of instructions executed within this time when running at a clock frequency f_{MCU} is $ED = t_{cmp} \cdot f_{MCU}$. We can now calculate

$$ED = f_{mcu} \cdot (C\frac{(V_{trig} - V_{min})}{I} - t_{chk}). \quad (4)$$

The MSP430-FR5737 datasheet [30] states that the current draw during the active mode at $1MHz$ goes from $200\mu A$ up to $420\mu A$, depending on cache hit ratio. A group of reasonable values for current consumption is $250\mu A$, $270\mu A$, and $310\mu A$, respectively corresponding to a 75%, 66%, and 50% of cache hit. From Hibernus++ [4] we know that t_{chk} is $1.4ms$ and V_{min} is $1.88V$. Fig. 8 shows the ED that we calculate according to the above derivations, which ranges from 2470 to 5800 instructions. We run our experiments using three representative values for ED: 3000, 4000, and 5000.

Measuring the baseline. LAYMAN-MEMORY must generate an independent test for any possible checkpoint location, which is at any line of code except the last one. For each of these potential checkpoint locations, LAYMAN-MEMORY must simulate a power failure at every instruction that follows the checkpoint within ED following instructions. As a result, the entire analysis for LAYMAN-MEMORY would require years to complete on a standard PC, as we argue earlier.

To obtain a quantitative baseline for comparison, we synthetically calculate the number of instructions executed by LAYMAN-MEMORY for a given benchmark as

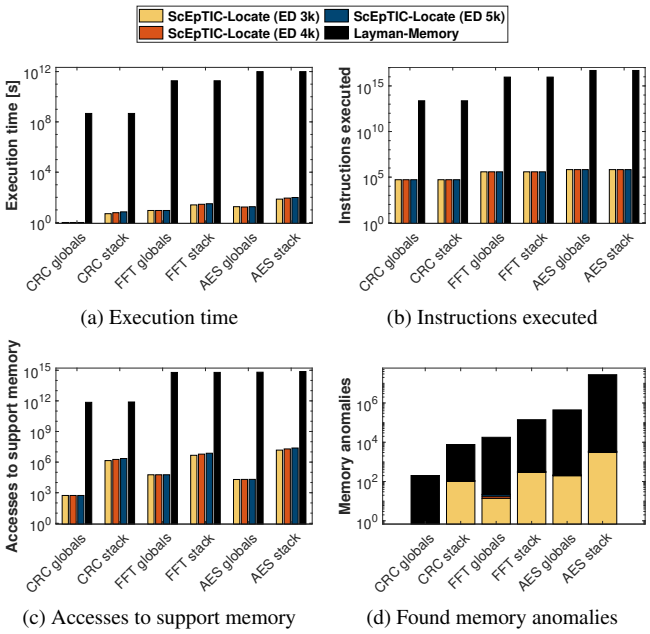
$$(\sum_{i=0}^{n_{ops}-1}(\sum_{j=i+1}^{n_{ops}}(j-i)+1)) - 1, \quad (5)$$

where i represents the checkpoints, j represents the power failures, and n_{ops} is the number of machine instructions in a sequential execution of the same code. The *execution time* for LAYMAN-MEMORY is consequently obtained by considering the emulation speed of SCEPTIC, which runs $5 \cdot 10^4$ instructions per second. With a similar reasoning, we also synthetically calculate the number of accesses to support memory and memory anomalies that LAYMAN-MEMORY finds.

6.2 Memory Anomalies: Results

A-priori scenario. Checkpoint calls are yet to be placed or we are employing reactive system support that potentially triggers checkpoints anywhere in the code. This configuration corresponds to the processing in Procedure 2, where SCEPTIC-EVALUATE stops at the first anomaly found in a given window of instructions, assuming that cascading effects of such anomaly are of no interest. The memory anomaly information that SCEPTIC-EVALUATE provides is thus equivalent to SCEPTIC-LOCATE; we only consider the latter.

Fig. 9 shows the results we obtain. Fig. 9a demonstrates that the execution time of SCEPTIC-LOCATE is 10 orders of magnitude lower than LAYMAN-MEMORY. In absolute terms, SCEPTIC-LOCATE constantly concludes the analysis in practical time across all benchmarks and memory configurations.



(a) Execution time (b) Instructions executed

(c) Accesses to support memory (d) Found memory anomalies

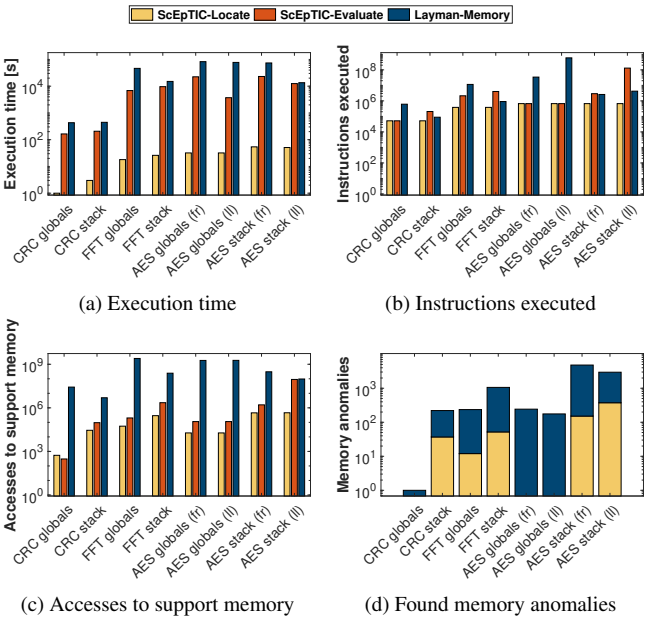
Fig. 9: SCEPTIC-LOCATE is orders of magnitutde faster than LAYMAN-MEMORY in the a-priori scenario. The X axis represents the benchmark and the memory slice on NVM.

Fig. 9a also shows that an increase of ED bears a minimal performance impact on SCEPTIC-LOCATE. The is explained in the results we obtain for the number of embedded code instructions executed and in the number of accesses to support memory, shown in Fig. 9b and Fig. 9c respectively. SCEPTIC-LOCATE executes the program sequentially to collect trace information and later analyzes a sliding window of instructions to locate memory anomalies. Increasing ED therefore does not increase the number of instructions that SCEPTIC-LOCATE executes overall, shown in Fig. 9b. It only slightly increases the accesses to support memory, shown in Fig. 9c.

Fig. 9d shows the number of found memory anomalies to analyze the output noise. LAYMAN-MEMORY returns information on many more memory anomalies due to the coarse-grained information it reasons upon. These anomalies are, however, semantically equivalent to those found by SCEPTIC-LOCATE. Programmers gain no insights from these additional information, which essentially represents a noisy output compared to the programmers' actual needs.

A-posteriori scenario. Fig. 10 shows the results for the *a-posteriori* scenario. For each benchmark, we place checkpoints accordingly to the loop-latch (ll) strategy of Mementos [32]. We also consider the function-return (fr) strategy for the AES benchmark, as it executes a significant number of function calls. Fig. 10a shows SCEPTIC-LOCATE with the lowest execution time. SCEPTIC-LOCATE is on average 3 orders of magnitude faster than LAYMAN-MEMORY and 2 orders of magnitude faster than SCEPTIC-EVALUATE. On the other hand, SCEPTIC-EVALUATE is on average 5 times faster than LAYMAN-MEMORY. LAYMAN-MEMORY has better performance for the AES benchmark as it finds memory anomalies earlier than SCEPTIC-EVALUATE due to the way the code is structured.

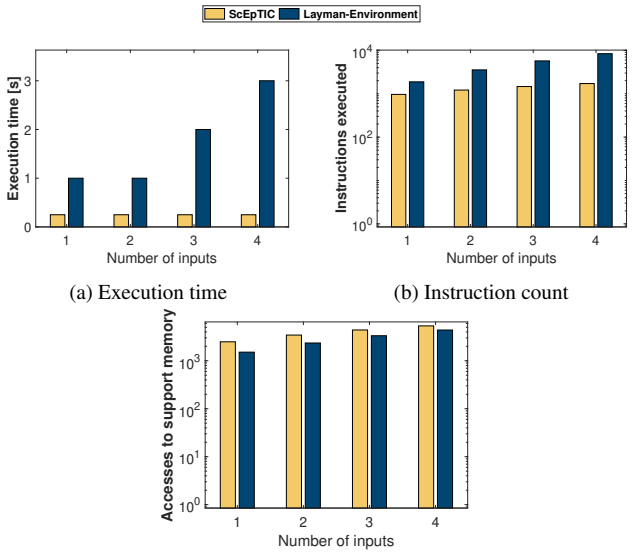
Emulating power failures requires SCEPTIC-EVALUATE and



(a) Execution time (b) Instructions executed

(c) Accesses to support memory (d) Found memory anomalies

Fig. 10: ScEpTIC-Locate is about 3 orders of magnitude faster than Layman-Memory in the a-posteriori scenario.



(a) Execution time (b) Instruction count

(c) Memory accesses

Fig. 11: ScEpTIC is 7x faster in determining the access semantics for environment input variables.

Layman-Memory to save a snapshot of the emulation state at every checkpoint. This is necessary to continue the analysis from a valid state, rather than an inconsistent one, in case any of the re-executed instructions yields a memory anomaly. This causes more accesses to the support memory, shown in Fig. 10c, that make ScEpTIC-Evaluate and Layman-Memory slower than ScEpTIC-Locate even when they execute the same number of instructions. This is the case for CRC and AES with global variables on NVM.

Fig. 10b also shows that when the stack is on NVM, ScEpTIC-Evaluate executes a higher number of instructions compared to Layman-Memory. This is expected, because Layman-Memory cannot pinpoint the instructions that cause a memory anomaly and the ones consuming the altered value. For this reason, it may not simulate power failures for executions where it already recognized a memory anomaly, even though there may be further memory anomalies in the same slice of execution that involve different instructions.

Despite the difference in the number of instructions executed, Fig. 10a shows that ScEpTIC-Evaluate is still faster than Layman-Memory. The reason for this is again in the accesses to support memory, as shown in Fig. 10c: ScEpTIC-Evaluate records only write events and then verifies when a read happens, whereas Layman-Memory compares the entire emulation state with a snapshot, resulting in higher overhead.

6.3 Input Access Analysis: Setup

We evaluate the performance of our analysis of input interactions with the environment, as explained in Sec. 4.1. As the analysis of output interactions uses almost identical techniques, as illustrated in Sec. 4.2, the conclusions we obtain also apply to that. In both cases, the actual procedure we apply is a limited variation of ScEpTIC-Locate.

Similar to Sec. 6.1, we employ a Layman-Environment baseline representative of how one would naturally operate to verify that, in the presence of arbitrary checkpoint operations and power failures, the behavior of a given input variable matches the intended semantics. In essence, this entails determining whether input variables behave according to a *long-term* or *most-recent* semantics against the re-executions of arbitrary code segments.

Benchmarks and configuration. The few works [18, 33, 26, 8, 37] considering intermittent executions together with environment interactions typically employ a typical sense-process-transmit application.

We consider four different input configurations: from one to four environment inputs through corresponding sensors. We use Mementos [32] again as checkpoint mechanism and rely on its manual strategy to keep checkpoint calls balanced with respect to calls to probe sensors. For example, with two inputs we place a checkpoint between their access calls, resulting in the sequence $read1() \rightarrow checkpoint() \rightarrow read2()$. This also corresponds to the approach that existing system support [18, 23, 11] employs to interleave calls to peripherals with checkpoints to ensure atomic execution of individual peripheral interactions.

As the procedure we apply is a variation of ScEpTIC-Locate and the baseline is, in fact, a variation of Layman-Memory, we use the same metrics as in Sec. 6.1. The number of found anomalies is, however, not applicable in this case.

6.4 Input Access Analysis: Results

Fig. 11 shows the results. Both ScEpTIC and Layman-Environment execute the benchmark within seconds, with ScEpTIC performing on average 7 times faster, as Fig. 11a shows. ScEpTIC takes the same time for the execution of the different input configurations, since it executes the program sequentially and requires no re-execution. Instead, the performance of Layman-Environment worsens with the number of inputs, since it needs to re-execute an increasing number of instructions as the number of inputs grows.

This performance reflects in Fig. 11b, where ScEpTIC is

shown to execute almost the same amount of instructions, independent of the number of inputs. The minor increase is merely due to separately processing different inputs. Instead, the number of instructions that LAYMAN-ENVIRONMENT executes increases with the number of inputs present in the benchmark, again because of the increase in the number of re-executed instructions with more inputs.

Fig. 11c accordingly shows that SCEPTIC accesses the support memory 1.4 times more than LAYMAN-ENVIRONMENT on average, as required by the processing applied to the relevant windows of instructions. The overhead that support memory accesses introduced in SCEPTIC is, however, significantly lower than the one of the actual re-execution of code segments in LAYMAN-ENVIRONMENT, ultimately resulting in faster executions for SCEPTIC.

7 Conclusion

Intermittent executions of battery-less embedded devices conceal hidden anomalies whose comprehensive treatment was not addressed by prior work. We fill this gap by investigating the anomalies occurring on memory and through environment interactions. Our contributions are made concrete in SCEPTIC, a code analysis tool for intermittent programs that uncovers previously unknown anomalies. Our evaluation indicates that SCEPTIC is *orders of magnitude* faster than the baselines we consider across a significant set of diverse benchmarks and configurations, enabling many types of analyses that would be otherwise impractical.

8 References

[1] S. Ahmed et al. The betrayal of constant power × time: Finding the missing joules of transiently-powered computers. In *Proceedings of LCTES*, 2019.

[2] A. R. Arreola et al. RESTOP: retaining external peripheral state in intermittently-powered sensor systems. *Sensors*, 2018.

[3] D. Balsamo et al. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters*, 2015.

[4] D. Balsamo et al. Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2016.

[5] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac. Sytare: a lightweight kernel for nvram-based transiently-powered systems. *IEEE Transactions on Computers*, 2018.

[6] N. A. Bhatti, M. H. Alizai, A. A. Syed, and L. Mottola. Energy harvesting and wireless transfer in sensor network applications: Concepts and experiences. *ACM Transactions on Sensor Networks*, 2016.

[7] N. A. Bhatti and L. Mottola. Harvos: Efficient code instrumentation for transiently-powered embedded sensing. In *Proceedings of IPSN*, 2017.

[8] A. Branco, L. Mottola, M. H. Alizai, and J. H. Siddiqui. Intermittent asynchronous peripheral operations. In *Proceedings of SenSys*, 2019.

[9] A. Colin, G. Harvey, B. Lucia, and A. P. Sample. An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems. *ACM SIGARCH Computer Architecture News*, 2016.

[10] A. Colin and B. Lucia. Chain: Tasks and channels for reliable intermittent programs. In *Proceedings of OOPSLA*, 2016.

[11] A. Colin and B. Lucia. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of CC*, 2018.

[12] A. Colin, E. Ruppel, and B. Lucia. A reconfigurable energy storage architecture for energy-harvesting devices. In *Proceedings of ASPLOS*, 2018.

[13] P. Derler, E. A. Lee, and A. S. Vincentelli. Modeling cyber–physical systems. *Proceedings of the IEEE*, 2011.

[14] M. Furlong, J. Hester, K. Storer, and J. Sorber. Realistic simulation for tiny batteryless sensors. In *Proceedings of ENSsys*, 2016.

[15] J. Hester, T. Scott, and J. Sorber. Ekho: Realistic and repeatable experimentation for tiny energy-harvesting sensors. In *Proceedings of SenSys*, 2014.

[16] J. Hester and J. Sorber. Flicker: Rapid prototyping for the batteryless internet-of-things. In *Proceedings of SenSys*, 2017.

[17] J. Hester and J. Sorber. The future of sensing is batteryless, intermittent, and awesome. In *Proceedings of SenSys*, 2017.

[18] J. Hester, K. Storer, and J. Sorber. Timely execution on intermittently powered batteryless sensors. In *Proceedings of SenSys*, 2017.

[19] M. Hicks. Mibench2 porting to IoT devices. https://github.com/impedimentToProgress/MiBench2, 2016 (last access: Jan 5th, 2021).

[20] H. Jayakumar, A. Raha, W. S. Lee, and V. Raghunathan. Quickrecall: A hw/sw approach for computing across power cycles in transiently powered computers. *ACM Journal on Emerging Technologies in Computing Systems*, 2015.

[21] H. Jayakumar, A. Raha, J. R. Stevens, and V. Raghunathan. Energy-aware memory mapping for hybrid fram-sram mcus in intermittently-powered iot devices. *ACM Trans. Embed. Comput. Syst.*, 2017.

[22] The LLVM compiler infrastructure. https://llvm.org/, 2003 (last access: Jan 5th, 2021).

[23] B. Lucia and B. Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of PLDI*, 2015.

[24] K. Maeng, A. Colin, and B. Lucia. Alpaca: Intermittent execution without checkpoints. *Proceedings of the ACM Programming Languages*, 2017.

[25] K. Maeng and B. Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *Proceedings of OSDI*, 2018.

[26] K. Maeng and B. Lucia. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *Proceedings of PLDI*, 2019.

[27] A. Maioli. Understanding and testing intermittence bugs in transiently-powered computers. Technical Report n 37/2019, Politecnico di Milano (Italy), 2019.

[28] A. Maioli. ScEpTIC documentation and source code. http://sceptic.neslab.it/, 2021 (last access: Jan 5th, 2021).

[29] A. Maioli, L. Mottola, M. H. Alizai, and J. H. Siddiqui. On intermittence bugs in the battery-less internet of things (WiP paper). In *Proceedings of LCTES*, 2019.

[30] Texas Instruments. MSP430-FR5737 datasheet. https://www.ti.com/lit/ds/symlink/msp430fr5737.pdf, 2017 (last access: Jan 5th, 2021).

[31] B. Ransford and B. Lucia. Nonvolatile memory is a broken time machine. In *Proceedings of MSPC*, 2014.

[32] B. Ransford, J. Sorber, and K. Fu. Mementos: System support for long-running computation on rfid-scale devices. *ACM SIGARCH Computer Architecture News*, 2011.

[33] E. Ruppel and B. Lucia. Transactional concurrency control for intermittent, energy-harvesting computing systems. In *Proceedings of PLDI*, 2019.

[34] E. Sardini and M. Serpelloni. Self-powered wireless sensor for air temperature and velocity measurements with energy harvesting capability. *IEEE Transactions on Instrumentation and Measurement*, 2011.

[35] E. Sazonov, H. Li, D. Curry, and P. Pillay. Self-powered sensors for monitoring of highway bridges. *IEEE Sensors Journal*, 2009.

[36] M. Spivak and S. Toledo. Storing a persistent transactional object heap on flash memory. In *Proceedings of LCTES*, 2006.

[37] M. Surbatovich, L. Jia, and B. Lucia. I/o dependent idempotence bugs in intermittent systems. *Proceedings of the ACM Programming Languages*, 2019.

[38] J. Van Der Woude and M. Hicks. Intermittent computation without hardware support or programmer intervention. In *Proceedings of OSDI*, 2016.

[39] K. Vijayaraghavan and R. Rajamani. Novel batteryless wireless sensor for traffic-flow measurement. *IEEE Transactions on Vehicular Technology*, 2010.

[40] J. Yang et al. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *Proceedings of SenSys*, 2007.