I/O Dependent Idempotence Bugs in Intermittent Systems

MILIJANA SURBATOVICH, Carnegie Mellon University, USA
LIMIN JIA, Carnegie Mellon University, USA
BRANDON LUCIA, Carnegie Mellon University, USA

Intermittently-powered, energy-harvesting devices operate on energy collected from their environment and must operate intermittently as energy is available. Runtime systems for such devices often rely on checkpoints or redo-logs to save execution state between power cycles, causing arbitrary code regions to re-execute on reboot. Any *non-idempotent* program behavior—behavior that can change on each execution—can lead to incorrect results.

This work investigates non-idempotent behavior caused by repeating I/O operations, not addressed by prior work. If such operations affect a control statement or address of a memory update, they can cause programs to take different paths or write to different memory locations on re-executions, resulting in inconsistent memory states. We provide the first characterization of input-dependent idempotence bugs and develop IBIS-S, a program analysis tool for detecting such bugs at compile time, and IBIS-D, a dynamic information flow tracker to detect bugs at runtime. These tools use taint propagation to determine the reach of input. IBIS-S searches for code patterns leading to inconsistent memory updates, while IBIS-D detects concrete memory inconsistencies. We evaluate IBIS on embedded system drivers and applications. IBIS can detect I/O-dependent idempotence bugs, giving few (IBIS-S) or no (IBIS-D) false positives and providing actionable bug reports. These bugs are common in sensor-driven applications and are not fixed by existing intermittent systems.

CCS Concepts: • **Computer systems organization** → *Reliability*.

Additional Key Words and Phrases: intermittent computing, energy harvesting

ACM Reference Format:
Milijana Surbatovich, Limin Jia, and Brandon Lucia. 2019. I/O Dependent Idempotence Bugs in Intermittent Systems. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 183 (October 2019), 31 pages. https://doi.org/10.1145/3360609

1 INTRODUCTION

Batteryless, energy harvesting technology allows devices to operate using energy collected from their environment such as tiny solar panels [Colin et al. 2018] or radio waves [Sample et al. 2008]. Energy harvesting frees devices from tethered power and avoids the environmental limitations and maintenance costs associated with batteries. Batteryless operation enables far-reaching sensor-attached applications, such as medical implants [Proteus Digital Health 2015], building and civil infrastructure monitors [Kim et al. 2007], and chip-scale satellites in space [Colin et al. 2018; Zac Manchester 2015]. A batteryless device typically buffers energy in a capacitor. A device's energy harvester operates with a power level that is often too weak to directly power a device. Instead, such a device operates only intermittently, in bursts, when energy is available in the energy buffer. After buffering a usefully large quantum of energy, the device powers on and activates its sensors,

Authors' addresses: Milijana Surbatovich, Carnegie Mellon University, USA, milijans@andrew.cmu.edu; Limin Jia, Carnegie Mellon University, USA, liminjia@cmu.edu; Brandon Lucia, Carnegie Mellon University, USA, blucia@cmu.edu.

2475-1421/2019/10-ART183
https://doi.org/10.1145/3360609

microprocessor (MCU), and radios to sense, compute, and communicate. Being active depletes the energy buffer and once it is empty the device powers off, erasing volatile state (e.g., registers, SRAM) and retaining non-volatile state (e.g., FRAM [TI Inc. 2017a], STT-MRAM [Guo et al. 2010]). Software on such a device executes intermittently, with operating periods punctuated by reboots. Several *intermittent execution models* ensure progress and memory consistency during an intermittent execution [Balsamo et al. 2016, 2015; Colin and Lucia 2016; Hester et al. 2017; Hicks 2017; Jayakumar et al. 2014; Lucia and Ransford 2015; Maeng et al. 2017; Ransford et al. 2011; Van Der Woude and Hicks 2016]. Checkpoints [Balsamo et al. 2015; Jayakumar et al. 2014; Lucia and Ransford 2015; Ransford et al. 2011; Van Der Woude and Hicks 2016] and tasks [Colin and Lucia 2016; Hester et al. 2017; Maeng et al. 2017] ensure that memory remains consistent and allow an execution to progress across power failures. While different in their details, these models operate by preserving an execution's context before power fails and restoring that context when power resumes, continuing the execution. After a failure, execution resumes at the last checkpoint or task boundary, and the execution model ensures that any re-executed memory updates produce the same final memory state: i.e., the re-execution should be *idempotent*.

While prior work has made advances in studying progress and memory issues, a program's interaction with mixed-volatility memory is not the only threat to correct and idempotent re-execution. A program's interactions with a device's environment also poses a threat to that program's correctness when executed intermittently. Applications of intermittent systems are typically deeply-embedded and driven by inputs from sensors and outputs to radios. To avoid the onerous and error-prone task of re-engineering millions of lines of low-level driver code, these input/output (I/O) operations should be moderated by existing sensor driver code. However, as we demonstrate in this work, the naive use of existing driver code in an intermittent system can cause a program to misbehave. If a region of code that interacts with a driver re-executes, the re-execution may *not* be idempotent, leading to incorrect intermittent execution behavior.

The goal of this work is to understand non-idempotent behavior caused by intermittent re-execution of input operations and develop techniques and tools to help programmers identify and fix bugs caused by such operations. We observe that the problem occurs after a system performs a repeated I/O operation in an intermittent execution. These repeated input operations (which we call "RIOs") may lead to memory inconsistency, failed subsequent input operations, incorrect results, or a crash. The key problem with a RIO is that the same input operation may produce different values in different intermittent re-executions. These I/O value differences can cause divergent control-flow decisions and divergent non-volatile memory updates, which is not idempotent and thus incorrect.

One of the contributions of this paper is to provide the first characterization of non-idempotent intermittent execution behavior caused by RIOs. Characterizing RIO bugs is critically important to the correctness of energy-harvesting systems. We show in Sections 7 & 8 that I/O device drivers from embedded systems libraries such as TI-RTOS [TI Inc. 2017b] are not always safe to use in an intermittent execution. The problem is especially dire because applications written for intermittent sensing platforms [Colin et al. 2018; Hester and Sorber 2017; Sample et al. 2008; Zhang et al. 2011a] typically rely heavily on I/O, and some systems, such as Wisent [Tan et al. 2016] and Stork [Aantjes et al. 2017] depend on I/O correctness to wirelessly reprogram RFIDs. These failures are even more serious for distributed intermittent devices, which recent work builds models of [Ma et al. 2018].

Another set of contributions of this work includes the algorithms and tools to help programmers to identify and fix problems in their programs. Our automated tool support for finding and fixing RIO bugs in intermittent systems will simplify intermittent system development significantly. We call this tool suite IBIS[1] . We observe that I/O-dependent idempotence violations fit a general code

[1] **I**/O **B**ug-finder for **I**dempotent **S**ections

pattern that is identifiable using a static program analysis. We develop a static analysis tool, IBIS-S, to detect and help programmers fix these bugs. IBIS-S asks the programmer to tag input operations and relies on a static taint analysis to find branching instructions dependent on I/O that might be involved in a violation. Leveraging the result of its taint analysis, IBIS-S looks for the specific RIO code pattern that may cause input-dependent idempotence violations. IBIS-S then provides the programmer with a detailed report of the involved code, data, and I/O. IBIS-S uses report filtering and validation to eliminate spurious reports, focusing programmer attention on the reports that are most likely to induce run-time errors. To avoid the conservatism and imprecision inherent in static compiler analysis, we develop a dynamic information flow tracking tool (IBIS-D) that identifies RIO bugs at runtime, as a program executes intermittently. IBIS-D's analysis dynamically tracks each operation that is control- or data-dependent on a tagged I/O operation. If such an input dependent operation non-idempotently manipulates non-volatile memory in subsequent re-executions, IBIS-D reports to the programmer the offending line of code and involved data.

We evaluate IBIS using 18 *real* programs containing code from the Texas Instruments Real-Time Operating System (TI-RTOS) and from prior literature [Colin and Lucia 2016; Maeng et al. 2017]. This evaluation shows that I/O-dependent idempotence violations are a real threat to porting existing code bases to intermittent systems, and that IBIS directs the programmer to these violations with very few (usually zero) false positives.

This paper makes the following contributions:

- The first characterization of RIOs and input-dependent idempotence violations.
- Algorithms for identifying code patterns that lead to input-dependent idempotence violations.
- A static tool (IBIS-S) for identifying and validating potential I/O-dependent idempotence violations.
- A dynamic information flow tracking tool (IBIS-D) for identifying actual I/O-dependent idempotence violations in real intermittent executions.
- An evaluation on widely used embedded OS and application code showing that IBIS effectively finds bugs with few or no false positives.

2 BACKGROUND AND MOTIVATING EXAMPLE

This work targets energy-harvesting devices that execute software intermittently [Balsamo et al. 2015; Lucia and Ransford 2015; Maeng et al. 2017; Van Der Woude and Hicks 2016]. Prior work studied how idempotent execution of code in such systems is complicated by Write-After-Read (WAR) dependences involving accesses to non-volatile data. We show using an example how RIOs also disrupt idempotence, causing input-dependent idempotence violations which cannot be straightforwardly fixed by existing checkpointing strategies. We also discuss how IBIS-S and IBIS-D can help programmers in combination with existing intermittent systems.

2.1 Computing in Energy Harvesting Devices

The Energy Harvesting Devices discussed in Section 1 buffer collected energy into fixed-size capacitors and run in short bursts followed by powered-off, recharging periods. Power failures erase volatile memory, e.g., stack, registers, and peripheral configurations, while preserving non-volatile memory. Intermittent applications rely on checkpoints [Balsamo et al. 2015; Jayakumar et al. 2014; Lucia and Ransford 2015; Mirhoseini et al. 2013; Ransford et al. 2011; Van Der Woude and Hicks 2016] or tasks [Colin and Lucia 2016; Hester et al. 2017; Maeng et al. 2017] to run to completion across power failures. A checkpoint may occur at an arbitrary point [Balsamo et al. 2015; Jayakumar et al. 2014; Ransford et al. 2011; Van Der Woude and Hicks 2016] in an execution. Task boundaries, which are like checkpoints, are inserted by the programmer at arbitrary code points [Colin and Lucia 2016; Hester et al. 2017; Lucia and Ransford 2015; Maeng et al. 2017]. In

this paper, we use the general term **boundary** to refer to both a checkpoint and task boundary. After a failure, execution resumes at the last boundary, the system ensuring memory consistency on restart. A key challenge is avoiding potential inconsistencies caused by problematic memory dependencies that lead to *non-idempotent* re-execution of some code, producing invalid results.

2.2 Non-idempotency Caused by WAR

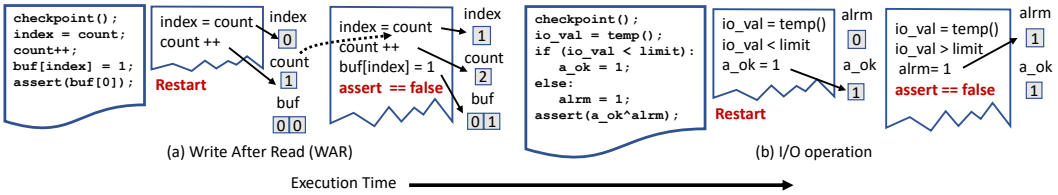


Fig. 1. Memory inconsistencies due to WARs and RIOs. A solid arrow indicates a memory update caused by an instruction. A dashed arrow indicates a data dependency from NV memory to a re-executed instruction.

Prior work observed that the intermittent re-execution of code containing a WAR dependence may not be idempotent; a read occurring after a power failure may consume a value from a WAR-dependent write that executed *before* the power failure. Figure 1 (a) shows how a WAR dependence can lead to memory inconsistency. The left box is `main()`. The global variable `count` is in non-volatile memory and initialized to zero. The middle box shows the first execution of `main`, which updates the `index` and `count` variables in non-volatile memory. Power fails before the buffer is updated. The right box shows a re-execution from a checkpoint at the first line of `main()`. The execution stores the wrong value of `count` into `index`. As a result, the code incorrectly updates the second entry of the buffer instead of updating the first. The assertion fails and the program crashes. The key to this failure is the WAR dependence between the operations manipulating the non-volatile variable `count`. A write (to `count`) occurs in the execution that the power failure interrupts. After the restart, the read of `count` consumes the value written before power failed, instead of its initial value. The problematic WAR dependence causes the re-execution of the code after the checkpoint to be non-idempotent, producing a different result from the first execution. WAR dependencies are a first and important cause of non-idempotent behavior that has been addressed by many prior efforts [Hicks 2017; Lucia and Ransford 2015; Maeng et al. 2017; Van Der Woude and Hicks 2016]. WAR dependencies, however, are not the only source of non-idempotent behavior.

2.3 Non-idempotency Caused by RIOs

Figure 1 (b) shows an example of how a RIO leads to incorrect behavior. The left box is `main()`, which reads from a temperature sensor. If the reading is less than some limit, the code sets `a_ok` to 1 and otherwise sets `alrm` to 1. At the end of the function, a sanity check asserts that only one of `a_ok` and `alrm` is set. The middle box shows an execution that reads a value from the sensor that is below the threshold, setting `a_ok` in non-volatile memory. Power fails and the program re-executes as shown in the right box. The code reads a sensor value that is above the threshold and sets `alrm` in non-volatile memory. At this point in the execution, both `alrm` and `a_ok` are set, which is incorrect. The assertion fails and the program crashes.

Part (b) shows that repeated invocations of the input operation—a RIO—produce different results, even without WAR dependencies (illustrated in Part (a)). It's clear that `alrm` and `a_ok` are write only. This means RIOs lead to the inconsistent memory state shown, even using a sophisticated checkpointing or task mechanism [Lucia and Ransford 2015; Maeng et al. 2017; Van Der Woude and Hicks 2016]. Each different result follows a different control-flow path in the program and updates

different non-volatile variables; the initial execution and its re-execution are *not* idempotent. This non-idempotence is not safe and yields an incorrect setting of `alrm` and `a_ok` that is not possible in any continuously-powered program execution.

The essence of the problem with the RIO in the figure is that different input values in intermittent re-executions of the same code region can lead to different branch outcomes and different non-volatile memory updates. These inconsistencies between consecutive re-executions are *input-dependent idempotence violations*, which we characterize for the first time in the context of intermittent systems in this paper. We provide a more precise account of how RIOs violate idempotency in Section 3.6.

2.4 Nuances in Fixing Idempotence Bugs Caused by RIO

Both checkpoint and task-based programming models ostensibly offer simple solutions to RIOs and input-dependent idempotence violations by placing a boundary after the I/O operation, ensuring the input is preserved in non-volatile memory. Re-execution from a boundary after the I/O will always follow the same path because each re-execution always uses the same I/O result. While this simple fix may be viable for some applications, it is simply inapplicable to others as it violates *timeliness* [Hester et al. 2017] and *input atomicity* [Colin et al. 2018; Kang et al. 2018], important properties pointed out in prior work.

Timeliness requires processing of an input value to occur within a fixed duration from its collection or else the input should be recollected. There is no bound on the duration that a device needs to recharge after a power failure, especially in the absence of predictable, harvestable energy. Placing a boundary between the I/O and its processing may break timeliness constraints. If an input is collected, execution crosses the boundary, and the power fails, the collected input will be arbitrarily old when execution resumes and the data is finally processed. Such a delay between input and processing may produce results that are inconsistent with reality.

Boundaries at arbitrary program locations may also violate the programmer's expectations about what executes *atomically*, violating key assumptions. For instance, a program may first check that a sensor is both powered on and listening on a serial bus before reading data from the sensor. A re-execution that starts from a boundary between the sensor check and the sensor read will *not* repeat the check before reading the sensor, leading to a failure to read the sensor.

Checkpoints and tasks do not preserve timeliness, so they are not a sure fix for RIOs. Similarly, addressing RIOs using checkpoints or tasks may interact badly with I/O. The absence of a simple fix to the RIO problem with existing execution models emphasizes the need for debugging support for input-dependent idempotence violations.

2.5 Workflow of Using Existing Intermittent Systems and IBIS' Use-case

To create intermittent applications, programmers have four main options: using a task-based runtime, using checkpoints that are annotated or automatically placed during compilation, using checkpoints that execute just in time while the program is running, or not using anything and ensuring correct execution by hand. Figure 2 shows the process involved to transform a correct C program into an intermittent program with each of these methods, as well as the intermittence bugs that will still remain. We indicate representative, but not exhaustive, examples of each system.

To use a task-based system, programmers must restructure the code into tasks that fit the system's language model. Task-based programs will not have timeliness (indicated by TIME in the Figure) or WAR idempotence bugs as programmers can control where execution restarts, and the runtime tracks and revert the effect of WAR accesses. I/O bugs will still exist as no existing task-based system includes the effect of I/O operations in its idempotence analyses.

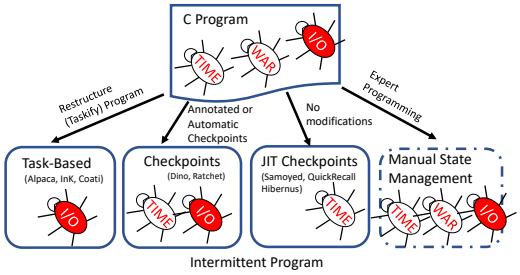


Fig. 2. The process of creating an intermittent program. I/O bugs are highlighted as they are the focus of this work. Manual state management is the most bug prone, but it is the most realistic option for low-level system code. In order, the works referenced are [Maeng et al. 2017] [Yildirim et al. 2018] [Ruppel and Lucia 2019] [Lucia and Ransford 2015] [Van Der Woude and Hicks 2016] [Maeng and Lucia 2019] [Jayakumar et al. 2014] [Balsamo et al. 2015]

To use a checkpoint-based system, programmers write the C code with little or no modification and expect the compiler to produce an intermittence-safe binary. If the programmer can control where the checkpoints are placed [Lucia and Ransford 2015], the programmer can avoid introducing timeliness bugs into the program. If checkpoints are determined by program analysis [Van Der Woude and Hicks 2016], the program will still be susceptible to them. As with task-based programs, these styles of checkpointing do not consider idempotence violations caused by I/O and will produce programs that still have I/O bugs.

Using a just-in-time (JIT) checkpointing system requires no programmer involvement, and the program will not have WAR or I/O bugs as code does not re-execute. Since the programmer cannot indicate which regions should execute atomically, there may still be timeliness bugs.

In the manual approach — which is particularly likely for the low-level driver and system code that IBIS targets — programmers do not use any runtime or compiler assistance but instead attempt to identify and remove all idempotence violations by hand. This process is tedious and error prone, much benefiting from debugging tools.

In summary, no current system option will guarantee a correct intermittent program, and most techniques—checkpoints that allow re-execution, tasks, and manual state management–can still result in I/O idempotence violations. IBIS-S' I/O idempotence analysis can be integrated into the WAR dependence analysis to remove I/O bugs for existing checkpoint and task-based systems, if freshness is not a concern. In all cases, IBIS identifies potential bugs caused by I/O, drastically reducing the programmer burden and providing valuable information towards fixing these bugs (See Section 6.2 for details).

3 SYSTEM OVERVIEW

We provide an overview of our tool IBIS, which is shown in Figure 3. IBIS consists of three main components: static taint analysis for identifying code patterns characteristic of input-dependent idempotence violations (Sections 3.3); bug validation that helps programmers determine if a reported bug causes an error in a software-emulated intermittent execution (Section 3.4); and a dynamic taint tracker that can detect or validate bugs in programs running on real energy harvesting hardware (Section 3.5). Next, we describe the input, output, and the design of each component. Detailed technical descriptions are in Sections 4, 5, and 6.1. Finally, we give a definition of the idempotence property that IBIS targets (Section 3.6).

3.1 Design Assumptions

We summarize the key design assumptions of IBIS: execution model and failure model.

Execution Model IBIS's static analysis implementation is not tied to any intermittent execution model. IBIS-S gives useful results whether the programmer is using checkpoints, tasks, or attempting to port code manually. IBIS-S provides two filtering modes which allow the programmer to refine the analysis if using checkpoints or tasks. The dynamic analysis tool IBIS-D is implemented for

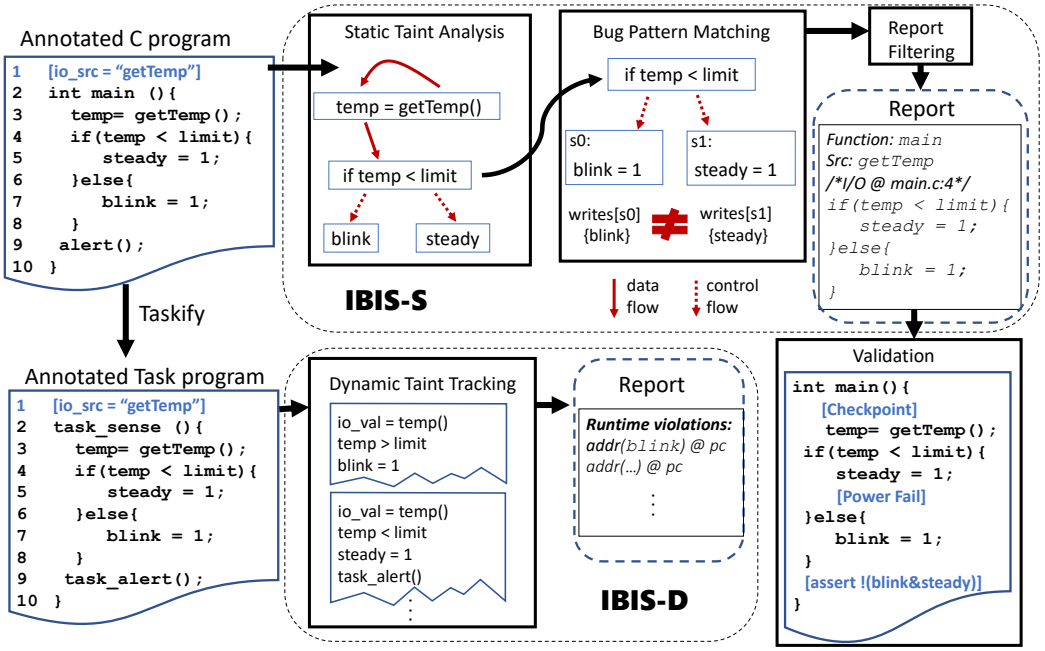


Fig. 3. High-level view of IBIS. The input to IBIS-S is programmer-annotated C programs. IBIS-S' bug identification sub-system uses a taint analysis and bug pattern matching. The bug reports are further filtered. Final bug reports can be validated using IBIS-S' validation sub-system which implements a checkpoint emulation system. IBIS-D runs on task-based programs and identifies concrete bugs at runtime.

programs using Alpaca, a task-based runtime. The dynamic analysis algorithm can be generally applied to other intermittent systems, not limited to task-based systems.

Failure Model We assume power can fail at any point, wiping the volatile state of the device, including the registers, stack, and peripheral state. On reboot, the device resumes execution from the last boundary point, restoring any checkpointed variables.

3.2 Programmer-supplied Input Annotations

The input to IBIS-S is a C program with annotations in the global region indicating which operations can produce an I/O input value. In Figure 3, the Input box shows an example annotated program, where line 1 in blue is an annotation stating that `getTemp` produces an I/O input. Radio receiver code, sensor drivers, and user interface components are common sources of input. Other operations may include driver interfaces exposed to the operating system or direct manipulation of memory-mapped I/O registers. Programmers can use IBIS-S' annotations to mark any of these operations as appropriate. Identifying input operations is unlikely to impose a high burden on the programmer because programmers are typically aware of points in code that interact with external devices, especially in an energy-constrained intermittent systems. As is typical in many embedded applications, IBIS-S assumes that input programs do not have recursive functions.

3.3 IBIS-S: Finding I/O Bugs Statically

Static Taint Analysis The heart of IBIS-S is a static taint analysis, implemented using the LLVM compiler framework, that identifies divergent control flow or memory updates depending on I/O

inputs. For the rest of this paper, we will call conditionals depending on I/O *tainted branches* or *input-dependent branches*. The sources (the initially tainted variables) of IBIS-S, taint analysis are input variables annotated by the programmer. For instance, in Figure 3, `temp` is a source. IBIS's taint analysis is an iterative, flow-sensitive, context-insensitive, inter-procedural algorithm. The analysis follows data and control flows to identify new tainted variables until a fixed-point is reached. In Figure 3, we use red arrows to illustrate dependencies that the taint analysis traverses. After the first iteration, the algorithm identifies line 4 as an input-dependent branch and also marks `blink` and `steady` as tainted.

Bug Pattern Detection IBIS-S uses the taint analysis results to find potential input-dependent idempotence violation bugs by identifying their characteristic code patterns. The first pattern, involving control flow, has two key features. First, buggy code must include a tainted branch. Second, the tainted branch must lead to *divergent write sets*, which are writes to different sets of non-volatile memory locations along different conditional paths. Two executions of programs with the above pattern could leave memory in an inconsistent state if the tainted branch resolves differently in two consecutive re-executions. If the write-sets in each execution diverge, the re-execution may not overwrite all locations written to in the first execution. As a result, updates from *both* paths from the branch remain in memory after the re-execution, which is incorrect. IBIS-S implements the bug pattern detection by combining an algorithm for identifying write sets with the taint analysis. As illustrated in Figure 3, IBIS-S reports a bug for the example.

A second pattern is when a pointer used to store to memory is tainted by I/O. If the value of the pointer changes on re-execution, the location being stored to changes, causing both updates to remain in memory even without branching control flow.

Bug Reporting and Report Filtering Once a bug pattern is identified, IBIS-S outputs a report. Each report lists the tainted branch, the variables in the branch's divergent write sets, and the sequence of taint propagation that led to the tainted branch. For instance, the report in Figure 3 states that in function `main`, the source of I/O input is `getTemp`, and an input-dependent branch is found at line 4 of the C file. The rest of the report is the branching instruction with only the write instructions to variables stored in non-volatile memory. IBIS-S can optionally filter reports to eliminate reported bugs that are unlikely to manifest divergent write sets.

A reported bug is not a true bug if, on re-execution, code preceding the tainted branch re-initializes all of the variables in the reported write sets. Re-initialization ensures that any re-execution is idempotent. Whether an execution re-initializes these variables depends on where a system resumes execution after a power failure. IBIS can be configured with two filtering heuristics based on common intermittent execution models: *task function filtering* and *I/O checkpoint filtering*. For our example here, the filter makes no difference. We will show an example of filtering in Section 4.3.

3.4 Failure Validation

IBIS provides support for programmers to understand and fix bugs. IBIS implements a report *validator* that executes the program in an emulated intermittent environment to determine whether the bug leads to a real failure. To use the validator, the programmer, using the bug report as a guide, specifies via annotations where to emulate a power failure, typically along one side of a tainted branch in the report. Next, using knowledge of the execution model, the programmer also specifies where to resume after the power failure, typically before an I/O operation or at the top of a tainted branch's function. Lines in blue in the validation box in Figure 3 are the annotations programmer would need to add. The last annotation is merely triggering an error in the execution. Finally, the programmer recompiles the annotated code, targeting a development machine. To generate input, programmers can either provide IBIS with trace files or a model of the input function

that generates random values fitting the input specification. The validator then runs the program with the programmer-provided inputs on the host machine, detecting crashes or assertion failures. Concrete executions from the validator reveal how a bug leads to a failure, helping to develop a fix.

3.5 IBIS-D: Finding I/O Bugs Dynamically

We implement a dynamic taint tracker, IBIS-D, that can detect I/O idempotence violations at runtime. At compile time the program is intrumented with calls to a runtime library. The runtime library tracks the taint of each address and gives a warning if the program reads I/O dependent values not consistent with the current execution. Consider an execution of `task_alert` in Figure 3. It updates the value of `blink` in memory and then restarts. IBIS-D tracks that `blink` is dependent on I/O, and once execution fails it records that the update was not committed. The subsequent re-execution takes the other path, updates `steady`, and transitions to the next task. IBIS-D tracks that `steady` is dependent on I/O and that it has been safely committed. Any uses of `steady` in `task_alert` will be safe, but any uses of `blink` will generate a runtime warning. Since bug reports are only generated when the tool detects an actual memory violation at runtime, IBIS-D has no false positives.

3.6 Correctness Criteria and Bug Definition

For a program with I/O operations to be correct on intermittent power, the result of a partial execution of a region with input i composed with a completed execution with input i' must be equivalent to a continuously powered execution with input i'. More concretely, let p be a program segment between two adjacent checkpointed boundaries (in a task-based system, p would be the code for a task). Note that the boundaries remove concerns of WAR dependencies. Let M denote non-volatile memory, which is a mapping from variables to values. Let $wt(i)$ denote the set of updates to non-volatile memory dependent on the input i (e.g., a sensor reading). Here, "dependent on" includes both data and control flow dependencies. Starting from an initial memory M_0, executing p until power failure results in the non-volatile memory being updated by writes dependent on i, which is written $M_0 \lhd wt(i)$. After reboot, p starts to execute on this updated memory. The re-execution reads a new I/O input i', writing to memory locations $wt(i')$. When p finishes, the resulting memory is $M_0 \lhd wt(i) \lhd wt(i')$. For a given input i', however, the final memory state for a *continuous* execution is $M_0 \lhd wt(i')$. If $wt(i)$ and $wt(i')$ do not write the same locations, these two states differ, and the idempotence property is violated.

The correctness criteria is $M_0 \lhd wt(i) \lhd wt(i') = M_0 \lhd wt(i')$ for any i and i'. The sufficient condition is $wt(i) = wt(i')$, and its negation is the input-dependent idempotency bugs that IBIS identifies. These are high-level informal definitions to illustrate the main points and omit final details such as sequences of I/O inputs and subset relations between $wt(i)$ and $wt(i')$. We leave a rigorous formal model of correct intermittent execution for future work.

4 BUG IDENTIFICATION AND REPORTING IN IBIS-S

We present the algorithms used by IBIS-S' bug identification and reporting. The order in which each component is called is slightly different from Figure 3. The top-level algorithm (shown in Algorithm 1) is a taint analysis. The bug pattern detection (shown in Algorithm 2) and filtering are called by the taint analysis eagerly during each iteration of the analysis. The program in Figure 4, extended from Figure 3, illustrates key points of the algorithms.

4.1 Top-level Taint Analysis

Algorithm 1 is IBIS-S' top-level algorithm: it traverses both local and inter-procedural dependencies to propagate taint information, identifies stores using tainted pointers, and checks tainted branches for the input-dependent idempotence violation bug pattern. The algorithm as implemented is not

sound – we discuss limitations in Section 4.4 and a precise formulation of the variable sets that a sound algorithm would collect in Section 4.5.

Taint analysis begins in TRAVERSE_DATAFLOW with the construction of an initial worklist containing only definitions at an I/O source, as annotated by the programmer (lines 2–5). Here *new_srcs* is the top-level work-list containing all instructions that are a new source of tainted input into any function. For instance, line 7 of the example program is in *new_srcs*. The main loop in the analysis iterates over items in the *new_srcs* (lines 6-32). The analysis then tracks which variables become tainted in a worklist, performing a depth-first search of dataflow along def-use chains. For each instruction in the *new_srcs*, the tainted variable in that instruction is added to a second function-local worklist *tainted_list*, which stores all the tainted variables within the current function being analyzed. For our example, *tainted_list* initially contains only `temp`.

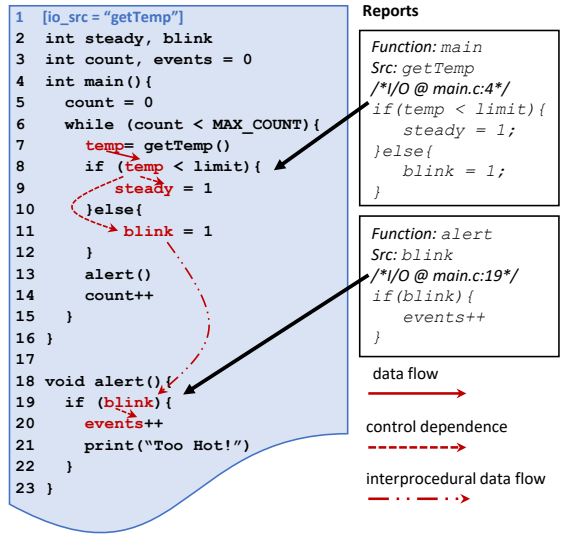


Fig. 4. This program samples from a temperature sensor, checks if the reading is above a certain level, and sets a flag. If the temperature is too high, it prints out an alert and adds to an event counter.

Then, all uses of the tainted variables are examined and categorized into three cases (lines 15–29). The first case is when a use of a tainted variable is in a definition of a variable. The defined variable becomes tainted and is added to the worklist *tainted_list* (lines 16–20). If the tainted operand in the definition instruction was the pointer operand, the calculated address may vary each execution, potentially accessing different memory. The analysis adds such a definition instruction to the bug report.

The second case is when the use is in a conditional of a branch instruction (lines 21–24). Here, the first feature of the bug pattern matches, and the analysis calls the CHECK function (Algorithm 2) to check for the match of the second feature of the bug pattern (details in Section 4.2). CHECK traverses the *taken* and *not-taken* control-flow paths that are dependent on the branch's outcome. The traversal also adds all variable definitions encountered on the taken and not-taken paths to the worklist being processed by TRAVERSE_DATAFLOW (line 20 in Algorithm 2). These definitions are tainted because they are control-dependent on the tainted branch. For instance, `steady` and `blink` will be added to the tainted list during this traversal.

In the third case, a use of the tainted variable is in another function and triggers inter-procedural taint propagation (lines 25–28). INTERPROC returns all the cross-function use sites. There are four operations that create such uses: (a) storing a tainted value to a global variable, (b) returning a tainted value, (c) storing a tainted value to a parameter passed by reference, and (d) a function call with a tainted variable as its argument. Figure 5 is a summary of the inter-procedural flows. In the case of any of these instructions, IBIS-S calculates instructions that are *sinks* (i.e. targets) of the taint propagation. For storing a tainted value to a global variable, all uses of the global variable in the whole program are sinks (will be tainted). For a return, the corresponding call instruction is the sink. For an assignment to a parameter passed by reference, the variables whose addresses are used by callers are sinks. For a function call with tainted arguments, all uses of the tainted arguments within

```
1: function TRAVERSE_DATAFLOW
2:   for all io_src ∈ uses(func_annotated) do
3:     new_srcs.add(io_src)
4:     ▷ init. new_srcs with annotated input func.
5:   end for
6:
7:   while !new_srcs.empty() do
8:     ▷ add to worklist if taint crosses funcs
9:     curr_val ← new_srcs.pop()
10:    tainted_list.add(curr_val)
11: ▷ traverse dataflow of curr_val within the func.
12:    while !tainted_list.empty() do
13:      var ← tainted_list.pop()
14:      for all use ∈ uses(var) do
15:        if is_def(use) then
16:          tainted_list.add(use.defvar)
17:          if is_ptr(var) then
18:        ▷ does this store use a tainted pointer?
19:            bug_report[func].add(use.defvar)
20:          end if
21:        else if is_branch(use) then
22:        ▷ does the branch match bug pattern?
23:          CHECK(use, tainted_list)
24:          FILTER(report)
25:        else if is_interproc(use) then
26:          sources ← INTERPROC(use)
27:    ▷ add interproc. flow to the work list
28:          new_srcs.add(srcs)
29:        end if
30:      end for
31:    end while
32:  end while
33: end function
```

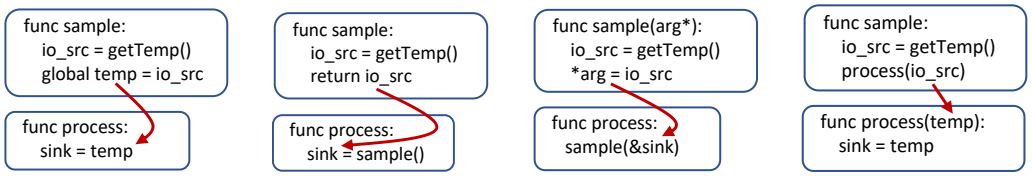
Algorithm 1. Taint Analysis

```
1: function CHECK(branch, tainted_list)
2:   brs ← successors(branch)
3:   for all b ∈ br do
4:     ▷ A fall-through has an empty write set
5:     if b == fallthrough then
6:       wrt[b] = empty; continue
7:     end if
8: ▷ find where the paths join
9:     join_blk ← PhiNode(branch).block
10: ▷ explore successors until join block
11:    blks_in_branch←
12:        GETBLKS(b, join_blk)
13:    for all blks ∈ blks_in_branch do
14:      for all insts ∈ blk do
15:        if inst == store then
16:          wrt[b].add(inst)
17:          tainted_list.add(inst)
18:        end if
19:        if inst == callinst then
20:          pre ← precomp_wr[callee]
21:          wrt[b].add(pre)
22:        end if
23:      end for
24:    end for
25:  end for
26:  for all i ∈ [0, brs.size − 1] do
27:    for all j ∈ [i + 1, brs.size − 1] do
28:      if wrt[brs[i]]!=wrt[brs[j]] then
29:        bugReport[branch].add(wrt[b])
30:        return true
31:      end if
32:    end for
33:  end for
34:  return false
35: end function
```

Algorithm 2. Checking for the bug pattern.

the callee are sinks. We used Steensgaard's alias analysis to track pass-by-reference parameters. To reduce extraneous may-alias reports, we modified it to only consider caller arguments with the same index as the tainted parameter. We discuss its limitations in Section 4.4.

The analysis adds all of these sinks of inter-procedural flow to the top-level worklist *new_srcs*, which will be processed by the outer-most loop of TRAVERSE_DATAFLOW. In our example the use of `blink` as indicated in Figure 4 triggers an inter-procedural taint propagation to a branch condition in `alert`. As a result, line 19 will be added as a new source. In the next iteration, `events` will become tainted because it is control dependent on `blink`.



(a) Tainted value is a global. Taint is propagated to any uses of the global.

(b) Tainted value is returned. Taint is propagated to caller.

(c) Tainted value stored in reference parameter. Taint is propagated to caller.

(d) Tainted value is used as function argument. Taint is propagated to callee.

Fig. 5. Summaries of inter-procedural taint propagation

4.2 Bug Pattern Detection

Besides reporting tainted pointers as bugs, IBIS-S uses the CHECK function listed in Algorithm 2 to determine whether a particular branch matches the bug pattern. The analysis examines the non-volatile variables written on all *taken* paths and all *non-taken* paths of the branch up to the point where the paths meet (i.e., at their common *phi* node in the CFG). If the taken paths write a set of variables that is not identical to those written by the non-taken paths, the analysis produces a bug report containing the branch, the taken and non-taken paths' write sets, and the taint propagation leading to the branch. IBIS-S treats fall-through paths as having an empty write set. In the example program, CHECK is called on the branches at lines 8 and 19. For the first branch, the target block on one side contains a write to `blink` at line 11, and the other side has a write to `steady` at line 9. For the second branch, the fall through is empty, and the taken side writes to `events` at line 20.

Inter-procedural Write Set Tracking. A function called along a path may write non-volatile state, and IBIS-S must track these writes when computing write sets. To avoid costly inter-procedural analysis, IBIS-S approximates a function's write set. The approximation pre-computes the *may-write* set for every function using a bottom-up traversal of the call graph (recall that IBIS-S assumes no recursive function calls). The traversal computes a summary of all writes each function may perform. The traversal adds a callee's write summary to its caller's write summary and terminates on reaching the top of the call graph. If CHECK encounters a function call while traversing a path, the pre-computed may-write summary of the function is added to that path's write set (lines 19-21 of Algorithm 2). Our write summaries are a conservative over-approximation (Section 4.4).

Branch Write Set Analysis Outcomes Three possible analysis outcomes of a branch are: (i) non-volatile variables written only on one side, (ii) *different* non-volatile variables written on different sides, and (iii) the *same* variables written on all sides. In a sound analysis, only the first two cases are potentially buggy. In the first case writes made on one side of the branch may be partially applied, but interrupted by a power failure. A later re-execution may traverse the other side of the branch, leaving an incorrect partial update. In the second case, execution may lead to a mixture of partial updates from one side of the branch (before a power failure) and from the other side (during re-execution), which is incorrect. In the third case, any writes applied in the first execution will get overwritten on subsequent executions no matter the value of the input, leaving the memory in a consistent state. The example in Figure 4 fits the second pattern as `blink` is written on one side of the branch and `steady` on the other.

4.3 Filtering Detected Bugs

IBIS-S filters reported bugs that are unlikely to lead to a failure. IBIS-S discards a report if the involved variables are *sanitized* through re-initialization when a re-execution starts, or if the dataflow necessary to trigger the failing execution is not possible on a targeted execution model.

IBIS-S' filtering mechanism is flexibly applicable to a variety of intermittent execution models, including task-based and checkpointing models.

A variable tainted by dependence on I/O is *sanitized* if it is overwritten at the start of any re-execution before any use of that variable. Figure 6 illustrates sanitization. The first execution follows the branch's taken path and writes to x, but the power fails before y is written. On re-execution, both x and y are set to zero, returning x and y to a valid initial state and avoiding a non-idempotent re-execution. IBIS-S uses a second dataflow analysis to identify operations that sanitize variables appearing in the initial bug report.

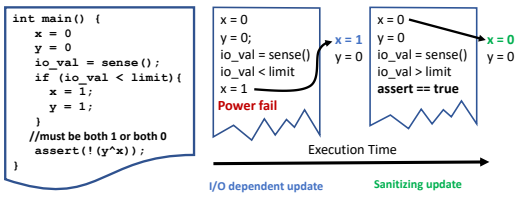


Fig. 6. x and y are control-dependent on a tainted branch but are always safely reinitialized.

The filtering algorithm can be configured to model re-execution after a power failure from either of two *restart points*: (i) *the top of the function* containing the report's tainted branch, or (ii) *immediately before the I/O operation* on which the report's tainted branch depends. Using the top of the function as the restart point corresponds to task-based execution [Colin and Lucia 2016; Maeng et al. 2017] and to checkpointing that places a checkpoint on function entry [Van Der Woude and Hicks 2016]. A restart point before an I/O operation corresponds to checkpointing at arbitrary locations at the last point that will preserve the operation's timeliness. IBIS-S' filtering algorithm examines instructions along any CFG path between the *restart point* and the report's tainted branch. If there is a write—a sanitizing write—to a variable in the report along *every* such path, the variable is removed from the report. If there is a sanitizing write to every variable in a report, the report is discarded.

Additionally, task-based systems or systems that checkpoint on function entry will only re-execute operations in the current function. If task function filtering is enabled, all reports whose I/O source and tainted branch were in different functions will be discarded. After task boundary crossing, the I/O dependencies are no longer non-idempotent, and all taints are cleared.

IBIS-S does not filter tainted pointer reports as whether the concrete value of the address changes on re-execution is information only available at runtime.

4.4 Soundness and Limitations

IBIS-S' over-approximation in computing may-write sets, context insensitivity, and heuristic-based alias analysis can cause false-negatives. Two may-write sets of a branch may match and thus not get reported, while concrete executions might have divergent writes. For example, in an if-else branch the "if" path might write variables x and y while the "else" contains a nested branch that writes x on one side and y on the other. The may-write sets of the top level branch are identical, x and y, but actually taking the "else" path will only write to one of the variables. Even though our analysis is flow-sensitive, plugging in may-write set summaries of nested branches brings unsoundness. At a glance, the may-write approximation seems to also cause false positives if a branch is always taken (or not taken). Consider the following program `if c x:= 3 else {if true x:= 3 else y:= 5}`. IBIS-S will report a bug, but any execution will only write to x. The underlying issue here, however, is that, like many static analyses, IBIS-S treats predicates as opaque and can't determine the value of the conditionals. Context insensitivity can cause similar false-negatives: if a function is called within the blocks dependent on a tainted branch, we add its precomputed may-write set to the result of the traversal.

When computing pass-by-reference data flows, we only check the caller argument with a matching index to the parameter. If the parameter aliases another argument, we would not explore that dataflow path, missing a potential bug. This does not cause false positives.

These tradeoffs were made as IBIS-S is for bug finding and its requirements for performance outweigh that for soundness.

4.5 Implementing a Sound Algorithm

The algorithms as implemented sometimes sacrifice precision for engineering and performance reasons. To be sound, IBIS-S would need both to calculate precise may and must-write sets and have precise taint tracking. Here we sketch a formulation of the precise sets of the variables that a tool like IBIS-S should track. While stating a semantic formulation is simple, implementing it in a static analysis remains hard.

A RIO bug occurs if a variable can be written on at least one path, but not all of them. If a variable were written to on all paths, it would always get sanitized on the current execution, and the inconsistent values would not remain in memory. We note then that the set to collect for a program region P is $MayWt(P) \setminus MustWt(P)$. This formulation must be quantified over the input values. For a variable to be in $MustWt(P)$, it must be written on all possible inputs, whereas to be in $MayWt(P)$ there must simply exist an input on which the variable is written. Implementing this formulation is complicated by what may and must writes sets mean under intermittence. Without any inputs, execution is deterministic. Even if there are multiple paths through a program, re-executions will always take the same path, so any written variables are effectively in the must-write set. This is why prior systems, which don't consider I/O, only checkpoint variables involved in WAR dependencies. Inputs introduce non-determinism, allowing the program to potentially take different paths each execution. Thus control-flow points that in no way depend on inputs are perfectly safe, whereas input-dependent conditionals require the $MayWt \setminus MustWt$ set to be calculated for code paths that are non-deterministically executed, i.e, the regions from the branch until control flow rejoins a deterministic path whose execution does not dependent on inputs.

This formulation is easy to state but non-trivial to implement. Computing conservative must-write and may-write sets is easy, but computing precise sets is difficult. A conservative must-write set may be too small, and a conservative may-write set may be too large. Since a sound tool needs the set difference between the must and may-write sets, if either set is not precise, the combination will not be precise either. Opaque path conditions can still cause errors for the static analysis as well, and the taint tracking itself must be sound to accurately determine the boundary between the deterministic and non-deterministic portions of the program region. The heuristic alias analysis we use has the potential to under-taint, possibly missing a code region that should be analyzed, but the unaltered Steensgaard's analysis has the potential to over-taint, which would cause an algorithm to consider deterministic paths as non-deterministic.

5 IBIS-D: DYNAMICALLY DETECTING RIO BUGS AT RUN TIME

We develop a dynamic taint tracking analysis, IBIS-D, to identify RIO bugs at run time as a program executes. Dynamic tracking overcomes key sources of imprecision in IBIS-S, namely heuristically guided pointer analysis and context insensitivity. The high-level analysis in IBIS-D is similar to the one in IBIS-S. The analysis tracks control and data dependencies from input operations and detects inconsistent writes to non-volatile memory. Unlike the static analysis used in IBIS-S, IBIS-D uses a compiler pass to insert instrumentation code into an application. The instrumentation runs code in IBIS-D's runtime library that tracks tainted variables and detects RIO bugs dynamically. The compiler pass inserts runtime calls at conditionals, I/O operations, and memory accesses. As the program executes, IBIS-D's runtime library keeps track of which variables are *tainted* because of a

control or data dependence on an input operation and whether these variables are *fresh*, written on the current execution, or *stale*, written on a prior, failed execution. To detect RIO bugs, the runtime propagates and analyzes data taint at each memory access. If a program loads a variable that is both tainted and stale, i.e., it became tainted before the most recent power failure, the variable's value is the result of a RIO bug. IBIS-D generates a bug report at each such load. We describe IBIS-D's taint tracking and monitoring algorithm and implementation in Section 5.1 and discuss the trade-offs between static and dynamic taint tracking and RIO bug detection in Section 5.2

5.1 Design and Implementation

IBIS-D's core algorithm implements dynamic taint tracking to track information-flow from I/O operations to other data in a program and freshness tracking to determine if a memory access is a bug. Data freshness is a key component in detecting intermittence bugs and is a major difference between IBIS-D and other dynamic taint tracking tools [Enck et al. 2010; Melicher et al. 2018]. Reading a fresh, tainted variable is safe because the input on which the variable depends was generated on the current execution. Reading a stale, tainted variable may represent a RIO bug because the input on which the tainted variable depends may not be consistent with the new input generated after the power failure. IBIS-D detects such accesses of stale, tainted data and generates a bug report including the variable's memory location and the program counter of the access. Taint is cleared from a variable when execution reaches a point where the input on which the variable depends will not re-execute if power fails (e.g., after taking a checkpoint or committing a task).

We prototyped IBIS-D for programs written in the Alpaca [Maeng et al. 2017] programming language and runtime. Alpaca uses a task-based execution model. Alpaca guarantees that the result of a memory update that is executed within one task is not visible to another task or a re-execution of the same task until the current task finishes and commits its updates to memory. The condition that IBIS-D monitors for — a read of a tainted variable written during a prior, partial task execution — breaks that guarantee and is a RIO bug.

IBIS-D explicitly maintains its taint-tracking metadata, maintaining two bits for each memory location: a *taint* bit and a *freshness* bit[2]. The *taint bit* for an memory location is set if the location's address is written by a memory access that is control or data dependent on I/O. IBIS-D clears the taint bit when execution reaches the end of the task and the update to the address is committed. The *freshness* bit for an address is set along with the taint bit on an input-dependent write. IBIS-D clears the freshness bit when the device reboots after a power failure. IBIS-D checks both bits before each memory access and generates a bug report if the taint bit is set and the freshness bit is cleared.

Taint propagation also occurs through registers and IBIS-D maintains a shadow register file with a taint bit for each register. IBIS-D sets the taint bit of a register if a tainted memory location is loaded into it and clears it if a non-tainted address is loaded. If a tainted register is used when updating a memory location, that memory location becomes tainted.

5.1.1 Instrumentation. IBIS-D's instrumenting compiler adds calls to its runtime library into the program's code. IBIS-D uses a front-end LLVM compiler pass to insert library calls at memory access instructions, control flow points, and calls to the annotated I/O functions. The front-end pass runs before register allocation, precluding taint propagation for registers in the front end. To identify registers through which taint propagates, IBIS-D has a second instrumentation phase after register allocation. This phase examines each instruction and adds taint propagation instrumentation that uses actual register names to manipulate the shadow register file that tracks taint.

[2]we track these bits for our 16-bit microcontroller's 16-bit address space

5.1.2 Runtime Library. The library provides APIs for memory access operations, control instructions, and calls to the annotated I/O functions. Calls to the library track freshness and taint propagation dynamically during program execution by analyzing and manipulating the metadata bits for each accessed memory location, detecting illegal uses of tainted variables written on previous executions.

Get_taint(addr, dst_reg) `Get_taint` is instrumented on loads and determines if the address to be loaded is currently tainted. If the taint bit of the address is set, then the function looks up the freshness bit. If the taint bit is set and the freshness bit is not set, the system generates a bug report; a program should not load a tainted value written by a previous execution attempt. The taint bit for the destination register of the load is set if the address was tainted and cleared if it was not.

Handle_control(type, reg_map) `handle_control` is instrumented on control flow instructions. The `type` parameter tells the function whether control branched or joined, and the `reg_map` parameter gives a list of the registers used in computing the branch condition, if one exists. If control branches and a register in the map is tainted, control flow dependent on I/O, and `handle_control` sets the *control-dependence flag*. We insert additional instrumentation at control-flow join points that clear the control-dependence flag.

Decide_taint(addr, reg_map) `Decide_taint` is instrumented on stores and determines if the destination address of the store should be marked as tainted. The parameter `reg_map` is a list of the registers used in computing the value to be stored. The taint bit for the address is set if the control-dependence flag is currently set or if any of the registers in the map are tainted. If the tainted bit is set, the freshness bit is also set, since the value is being written on the current execution.

Set_taint(ret_reg) `Set_taint` is instrumented on calls to the annotated I/O functions, and sets the taint bit of the return register of the call.

On_reboot and on_transition After transitioning between tasks, non-idempotent I/O values written in the current task cannot change and are safe to use on subsequent execution attempts. Immediately after the current task pointer is updated, `on_transition` clears the taint bit of any addresses written by the completed task. After rebooting, no writes are fresh, so `on_reboot` clears all the freshness bits.

5.1.3 Report to the Programmer. IBIS-D detects violations of the correctness invariant — accesses to tainted data written on a prior execution — as they occur and outputs the address of the loaded variable and the program counter. The programmer can use these to find the variable name and line number of the violating read in the source code. Tracking and outputting more data, such as the original taint source and branch like IBIS-S does, would incur costly runtime and memory overheads. IBIS targets embedded devices that typically lack displays. Our implementation produces reports over a serial UART link connected via USB to a workstation machine.

5.2 Static versus Dynamic Taint Tracking

IBIS-S and IBIS-D provide complementary benefits to the programmer. IBIS-S can quickly analyze the entire program and provide detailed bug reports. Reporting the I/O source of the bug, the tainted branch, and all potential non-idempotent writes helps the programmer reason about and fix the root cause of the bug. As discussed in Section 4.4, IBIS-S can produce false positive reports and may not explore all tainted code paths, especially if there is complicated aliasing. By contrast, IBIS-D will not produce false positives, generating a report at an actual violating access only. As with any dynamic analysis tool, IBIS-D produces no false negatives *if* IBIS-D is able to explore all possible executions. To span the space of intermittent executions requires spanning the entire space of data inputs to the program, and also exploring all possible points at which a power failure may occur. Thoroughly exploring this space is a compelling topic for future intermittent systems research.

While dynamic analysis does have benefits, particularly in accuracy, dynamic tools that instrument code have an inherent downside on intermittent systems. For task-based models or statically placed checkpoints, the code in a task or between checkpoints has to be able to complete with one buffer of energy to ensure forward progress. Estimating how much energy is needed for a region of code to run is non trivial, and there are compiler tools to help the programmer write programs that will complete [Colin and Lucia 2018]. Even if the programmer wrote a correct program that will always finish, instrumenting a program with more runtime calls will increase the size of the tasks or checkpointed regions and can break forward progress. This problem is not particular to IBIS-D but is an outstanding problem for any instrumentation tools targeting intermittent systems. Verifying that a transformation on an energy-safe program results in a program that is still energy-safe is also a topic for future research.

6 SOFTWARE-BASED VALIDATION AND BUG FIXING

IBIS helps the programmer validate a reported bug by running the program in a software-based intermittent execution emulator, enabling the programmer to directly observe the symptoms of a failure caused by the bug. IBIS-S' analysis can be integrated into existing runtimes to enable the runtime to fix bugs with a little programmer effort, or it can provide the programmer with enough information to reason about and fix the bug manually. We describe details of the validation system in Section 6.1 and recommendations on using IBIS to assist in fixing bugs in differing runtime systems in Section 6.2

6.1 Validation Sub-system

IBIS's intermittent execution emulator allows the programmer to explicitly specify the in-code locations of checkpoints and power-failure emulation points. The emulator also allows the programmer to mark data as residing in either non-volatile or volatile memory and to mark which data to include in a checkpoint. The emulator executes the program on inputs generated using a function provided by the programmer, and captures execution context (i.e., registers and stack) and explicitly marked state at each checkpoint. When the emulator encounters a power-failure emulation point, execution returns to the execution context set by the most recently executed checkpoint, restoring all checkpointed data. The programmer can systematically or randomly place checkpoints and power failure emulation points in their code and run in the emulator repeatedly. They can provide random inputs or real traces to the emulator. If an execution in the emulator causes a crash, incorrect output, or assertion failure, the programmer can study the emulated execution to better understand the failure, IBIS's bug report, and the bug fix.

We implemented the emulator in C++ to be portable. The implementations of power failure emulation, checkpointing, and restart all rely on `setjump` and `longjump`. `setjump` emulates checkpointing by saving the execution context from the point of its invocation to a *jump buffer*. `longjump` takes a jump buffer as a parameter and reverts execution to the point specified by the buffer, emulating an intermittent power failure that resumes at the checkpoint captured by `setjump`. A variable updated between a `setjump` (checkpoint) and `longjump` (restart) retains its value, correctly emulating the reboot behavior of non-volatile memory. To emulate variables in volatile memory and checkpointed data in non-volatile memory, the emulator provides an access macro. For data accessed with this macro, IBIS maintains an *emulated volatile array*, a key-value store that maps addresses to their current value. The emulated volatile array maintains a checkpointed value for each entry. At a checkpoint, the emulator copies each entry's current value into its checkpointed value. At a power failure, the emulator resets each entry's current value to its checkpointed value.

6.2 Fixing Bugs with IBIS

IBIS-S can be used to guide fixing bugs as well as detecting them, both through programmer intervention and augmenting existing intermittent runtimes. IBIS-D gives the programmer information about where the program fails due to bugs reported by IBIS-S, but it does not provide as much information about the root cause of the bug.

Tasks IBIS-S can be directly integrated into systems that use WAR analysis to detect and back-up potentially inconsistent variables. IBIS-S' taint tracking analysis can be combined with the existing program analyses. The runtime mechanisms will then correctly restore the memory locations updated non-idempotently through I/O dependencies. Any false positives that IBIS-S reports will not make the runtime behave incorrectly, merely conservatively. This solution enables existing systems to maintain both memory consistency and timeliness in the face of RIOs. Repeated I/O operations can execute consistently without any task or checkpoint boundaries between the I/O operation and the dependent branch. Note that this process is not fully automatic as the programmer still needs to annotate each program with the input sources for IBIS' analysis to run.

Checkpoints Checkpoint systems that back-up and restore variables [Lucia and Ransford 2015] can be automatically fixed while preserving timeliness in the same manner as task-based systems. Checkpoint systems that try to decompose programs into idempotent regions [Van Der Woude and Hicks 2016] can use IBIS-S' analysis to preserve memory consistency, however placing checkpoints between I/O reads and uses of the I/O will break timeliness.

Manual Fixing IBIS' reports provide programmers with with the specific code region and variables that can become inconsistent, simplifying the debugging process. Depending on the timeliness and atomicity constraints on the application, programmers can use this information to reinitialize or back-up any critical variables, or to ensure that the I/O does not re-execute. IBIS can help programmers determine possible root causes of the bug, but it is a bug detection tool, not a debugging infrastructure.

7 EVALUATION

We evaluate IBIS on whether IBIS-S and IBIS-D can find input-dependent idempotence bugs in real applications and do so efficiently and accurately. An associated goal is to investigate whether these bugs appear in real applications using existing intermittent systems. There is no standard benchmark suite for intermittent devices, so this evaluation assembles a collection of embedded system code taken from several sources, including the Texas Instruments Real Time Operating System (TI-RTOS) sensor drivers and application libraries [TI Inc. 2017b], and applications from the literature [Colin and Lucia 2016; Maeng et al. 2017] obtained from the authors. Our evaluation of the accuracy of IBIS-S' analysis categorizes bug reports as true or false positives, showing that IBIS-S reports very few or no false positives. IBIS-D is limited to benchmarks written in the Alpaca language. We run the tool on Alpaca apps created from the TI-RTOS sensor drivers and the Alpaca applications obtained from the authors. We evaluate IBIS-D on whether it detects the bugs that IBIS-S reports as bugs (the true bug column in Table 2) and whether the dynamic taint tracking overheads are reasonable for a development phase tool. We analyze the bug detection capability of IBIS-D in relation to IBIS-S' reports as techniques to fully explore the input space of programs running intermittently is an area of future work.

Our evaluation shows that bugs due to input-dependent memory updates do occur on real applications and will not be fixed by existing systems. IBIS detects them efficiently and accurately.

7.1 IBIS-S Benchmarks

Table 1. Application Characteristics. The origins (in order): [Colin and Lucia 2016; Colin et al. 2018; Maeng et al. 2017; Sample et al. 2008; TI Inc. 2017b]

Origin	App	LoC	#Func.	#Branch	IO Ops.
TI-RTOS	mpu	1136	43	227	37
	hdc	256	14	33	2
	bmp	363	12	72	11
	opt	240	14	38	4
	tmp	286	12	54	10
	wsn	342	22	34	1
	easylink	1120	36	267	10
WISP	rfid	478	11	49	2
Capy	prox	606	39	161	8
Chain, Alpaca	gesture	275	22	30	2
	hmc	223	30	22	2
	tempalarm	296	20	51	1
	ar	530	20	107	1
	bc	479	16	98	1
	blowfish	509	23	113	1
	cem	413	23	62	1
	cuckoo	533	36	84	2
	rsa	887	28	217	2

IBIS-S is evaluated on code written for real embedded devices, summarized in Table 1. We included code from TI-RTOS drivers and application libraries [TI Inc. 2017b], WISP core library code [Sample et al. 2008], Capybara sensor driver and application code [Colin et al. 2018], and intermittent application benchmarks [Colin and Lucia 2016; Maeng et al. 2017]. The TI-RTOS code was not written for an intermittent programming model, but we include them to show the difficulty of porting embedded-C to intermittent systems and to stand in for checkpointed code. Checkpointing systems are intended to work on C code with little or no modifications as discussed in Section 2.5, and programmers trying to port code-bases manually will also start from unmodified existing C code. We included the drivers for motion processing (`mpu`), humidity (`hdc`), pressure (`bmp`), optical (`opt`), and thermopile (`tmp`) sensors. We also studied a library and application built using the TI-RTOS RF communications driver: `easylink`, a high-level radio API, and `wsn`, a wireless sensor data aggregator. To analyze these programs on our development machines, we abstracted or removed system calls or device specific pragmas irrelevant to IBIS-S' analysis. We assume that all variables are stored in non-volatile memory. From `libwispbase` [Sample et al. 2008] we included `rfid`, an RFID EPC Gen2 [EPCglobal Inc. 2015] decoder driver. We included `prox`, which is Capybara's [Colin et al. 2018] driver for the APDS9600 proximity and gesture sensor. An important characteristic of these low-level code bases is that porting them to use tasks [Colin and Lucia 2016; Maeng et al. 2017] or checkpoints [Balsamo et al. 2015; Jayakumar et al. 2014; Van Der Woude and Hicks 2016] would be extremely difficult because of subtle timing conditions and environmental interactions. The difficulty of using tasks or checkpoints to handle potential idempotence violations increases the value of IBIS.

We also analyzed applications from prior work written using task-based intermittent systems [Colin and Lucia 2016; Maeng et al. 2017]. We analyzed three applications using Chain [Colin and Lucia 2016]: `gesture`, `hmc`, and `tempalarm` ; and six applications using Alpaca [Maeng et al. 2017]: activity recognition `ar`, bit count `bc`, `blowfish`, a compression program `cem`, `cuckoo`, and `rsa` encryption.

Table 1 shows characteristics of our benchmarks. The programs have an average of 498 lines of code, ranging from 223 to 1136 lines. The average function count is 23 with a high of 43, and the average number of branch instructions is 95. The majority of programs have one or two distinct I/O operations, with the highest number being 37, in `mpu`.

7.2 IBIS-S Bug Detection Efficacy and Efficiency

The main results of our evaluation are summarized in Table 2. The first column is the number of total reported bugs before filtering. Next columns are: the runtime in milliseconds, the number of total reports, two types false positive reports (**NoBg** and **NoUse**), and true positive reports for each filtering strategy (task and I/O point). We elaborate on the the false positives in Section 7.3.

IBIS-S is efficient: the run times for both filtering methods, shown in Table 2, are less than 0.1 seconds for all cases except `prox`. `Prox` takes longer partly because it has a high count of total reports (31), most of which get filtered out. We note that the largest benchmark, `mpu`, has a low runtime of 13.1 ms. Additionally, we eschewed implementation choices that can make static analysis scale poorly, such as context-sensitive analysis and expensive alias analysis.

Table 2. Categories of false positives and true bugs based on filtering point, runtime in milliseconds.

	NoFlt	Task Filter					I/O Filter					
App	**Tot.**	**R(ms)**	**Tot.**	**NoBg**	**NoUse**	**Bug**	**R(ms)**	**Tot.**	**NoBg**	**NoUse**	**Bug**	**Vld.**
mpu	17	8.5	5	1	1	3	13.1	16	4	5	7	S
hdc	2	2.3	2	0	1	0	3.5	2	0	1	1	S
bmp	3	6.3	1	0	1	0	9.7	1	0	1	0	n/a
opt	3	2.4	1	0	0	1	3.5	1	0	0	1	S
tmp	3	5.8	3	0	1	2	8.7	3	0	1	2	S
wsn	6	3.7	1	0	0	1	4.5	3	0	0	3	S
elink	6	6.1	1	0	0	1	7.5	6	0	0	6	S
rfid	6	4.4	0	0	0	0	5.3	6	1	0	5	S
prox	31	165.9	4	0	0	4	231.8	10	3	2	5	S
gest	2	2.1	2	0	0	2	3.0	2	0	0	2	S
hmc	1	2.2	0	0	0	0	2.9	1	0	0	1	S
temp	0	1.5	0	0	0	0	2.2	0	0	0	0	n/a
ar	0	2.3	0	0	0	0	2.2	0	0	0	0	n/a
bc	0	19.6	0	0	0	0	25.2	0	0	0	0	n/a
bfish	0	1.7	0	0	0	0	2.5	0	0	0	0	n/a
cem	0	1.9	0	0	0	0	2.5	0	0	0	0	n/a
ckoo	0	11.4	0	0	0	0	16.2	0	0	0	0	n/a
rsa	0	7.9	0	0	0	0	11.8	0	0	0	0	n/a

The data show that the majority of OS, driver, and library code have at least one true bug, ranging from zero (`bmp`) to seven (`mpu`). The presence of these bugs demonstrates the need for IBIS in bringing important I/O support code to intermittent systems. Using existing systems to port delicate, timing-sensitive OS-level code to intermittent systems will likely result in subtle I/O related bugs. Programmers need IBIS's help to detect these bugs and to reason directly about the effect of I/O and input-dependent idempotence violations. In contrast, there are zero reports across all but two of the task-based programs. The absence of bugs in these programs is unsurprising for several reasons: the programs were written with tasks in mind, the programs do not make heavy use of I/O, and the programs have intervening task boundaries between gathering the I/O and any branches off of I/O dependent values. Table 2 also shows that filtering is effective at eliminating reports for both strategies (more details in Section 7.4).

IBIS-S reports bugs with high accuracy, producing very few false positives (the difference between column **bug** and **Tot.**, or the sum of column **NoBg** and **NoUse**) after filtering. There are very few false positives across applications, regardless of which filtering strategy IBIS uses. Many cases have zero false positives, a few cases have just one or two, and `mpu` has the most (details in Section 7.3).

The last column shows the validation results. If all of a test case's reported bugs were validated using IBIS's software validator (Section 6.1) the column contains an 'S'.

We did not report bugs found by IBIS-S to the developers, as many benchmarks are not currently designed to run on intermittent systems and these bugs only manifest in the intermittent context. IBIS is designed for future developers who want to target intermittent systems.

7.3 False Positives Classification

False positives are further broken down into two types: *not-a-bug* (**NoBg**) or *no-use* (**NoUse**). The first type simply does not correspond to a bug: the reported input-dependent idempotence violation does not lead to a failure. IBIS-S produced such reports for several of the sensor benchmarks: eight total, with four and three for `mag` and `prox` respectively and one for `rfid`. Not-a-bug reports occur

because reported operations do not leave state inconsistent or a failing execution is infeasible. For example, in `mpu` IBIS-S's analysis identifies a switch statement with a fall-through case that has a different write set from the other cases and could lead to an idempotence violation. This switch statement cannot lead to a failure, however, because an (implicit) constraint on the switch condition variable prevents the fall-through from executing, making the failing execution infeasible.

The second type occurs when the program exhibits the bug pattern but the tainted data are not used. For instance, in `mpu`, a variable *val* is both defined and used in the taken branch of an if statement, but not used elsewhere. IBIS-S generates a bug report because *val* could be updated in only one of the execution paths. However, since *val* is not used, an inconsistent write to it will not manifest as a bug. IBIS generates such reports for several of the sensor driver benchmarks.

7.4 Effect of Filtering Reported Bugs

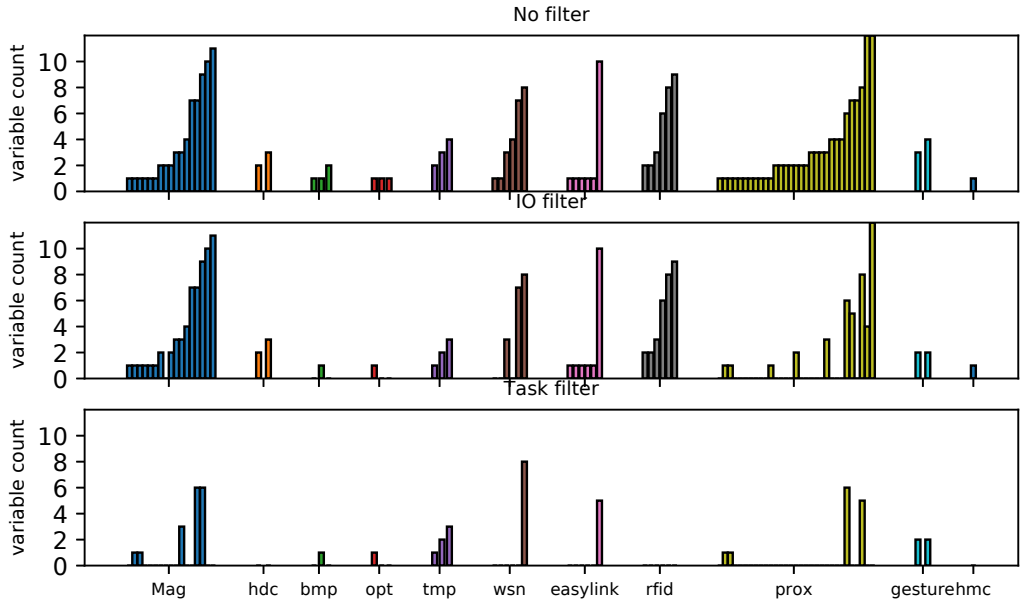


Fig. 7. The difference in bug reports and variable count depending on filtering point

Recall that filtering eliminates variables from a report by identifying sanitizing writes to those variables; if filtering eliminates all variables from a report, the report itself is eliminated. The most significant reduction is in `prox`, where moving from no filtering to task filtering reduces the total number of reports from 31 to 4. Other applications (`rfid` and `easylink`) see significant decreases (from 6 to 0 and from 6 to 1 respectively). These cases have sanitizing writes to tainted variables occurring between the top of the function and the reported branch or the original I/O source and the tainted branch are in different functions, making the I/O flow safe but untimely. I/O filtering sees similar decreases in reports in many cases, although in some cases it eliminates fewer reports than ToF filtering. For example, I/O filtering does not reduce the number of reports for `easylink`, but task filtering removes all but one report.

While Table 2 shows that filtering is effective at eliminating reports, filtering also *simplifies* remaining reports by eliminating sanitized variables from those reports. Figure 7 plots one bar per report, with each bar indicating the number of variables in its report. The top plot is no filtering,

the middle is I/O filtering, and the bottom is task filtering. The data show that filtering reduces the number of reports – there are fewer total bars in the plots for the filtering techniques– and that filtering simplifies reports – some bars are shorter in the plots for the filtering techniques (e.g., reports for `tmp`, `easylink`, and `mag`). By simplifying reports, IBIS-S makes debugging simpler for programmers.

Overall, task filtering is more effective than I/O filtering because the former permits a longer span of code in which sanitizing writes may occur. This also explains why more true bugs exist with I/O filtering than task filtering for `mpu`, `easylink`, and `hmc`.

7.5 IBIS-D Benchmarks

To detect bugs using IBIS-D, we needed to make the benchmarks execute on actual energy harvesting hardware. To create applications from the TI-RTOS benchmarks, we extracted the buggy functions and turned them in to runnable applications using the Alpaca programming language and runtime [Maeng et al. 2017]. We first extract all functions that contained a true bug with task-filtering. Each function is then turned into a task and the tasks are connected into a logical application. This procedure is necessary as most of the buggy programs were drivers whose functions do not interact directly with each other. For example, `mpu` had bugs in a function that initializes the calibration data and a function that reads the magnetometer, using the calibration data to scale the values. In normal use, these would both be called by a higher-level application that is using the sensor. The extracted app changes the two functions to tasks and connects them, adding a task that uses the result of the read. The goal of the bug extraction is to preserve the key control and data dependencies surrounding the bug pattern. To measure the overheads on realistic workloads, we also run the tool over the Alpaca applications we obtained from the authors. We show a summary of the application in Table 3. Apps that were created through bug extraction are marked with the superscript ^{x}. The two Alpaca apps marked with an asterisk could not run on our system due to device memory limitations.

Table 3. Summary of apps using the extracted code

App	Extracted Functions	App Description	LoC
mpu^x	Calibration Init, MagRead	App initializes the calibration, reads data, uses the data	233
hdc^x	HdcRead	App reads data, uses it	145
bmp^x	BmpRead, BmpConvert	App reads raw data and converts it to useable format	263
opt^x	OptRead	App reads data, uses it	161
tmp^x	TmpRead	App reads data, uses it	163
wsn^x	ProcessLoop, ReceivePacket, UpdateNode	App receives a packet, processes it, and adds it to a node list	249
elink^x	ReceivePacket	App receives a packet and processes it.	309
ar	Entire App	Activity Recognition	530
bc	Entire App	Bit Count	496
cem	Entire App	Compression App	429
cuckoo	Entire App	Cuckoo Hashing	533
rsa*	–	–	–
blowfish*	–	–	–

7.6 IBIS-D Bug Detection Efficacy and Efficiency

We summarize IBIS-D's performance in Figure 8, which shows runtimes, and Table 5, which shows a breakdown of the instrumented calls and bug detection results.

To gather run time information, we ran instrumented and unaltered versions of the applications on both continuous and intermittent power. Gathering on continuous power gives the slowdown

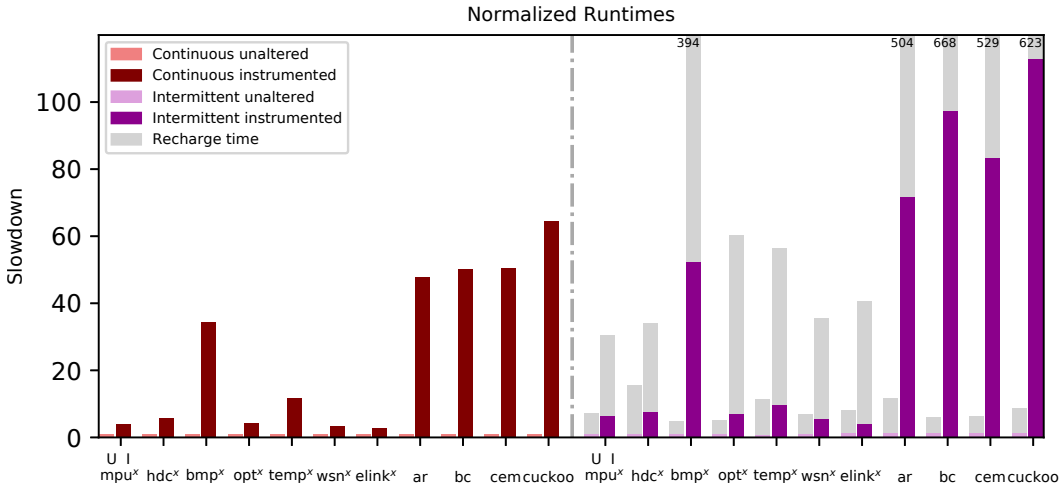


Fig. 8. Normalized runtime of instrumented and unaltered apps on both continuous and intermittent power with identical inputs

directly caused by the instrumented runtime library, as it factors out the recharge time and the code re-execution that occurs when the device runs intermittently. IBIS-D has an average of 25x slowdown, which is in line with the slowdowns produced by other memory checking tools such as Memcheck or TaintCheck [Nethercote and Seward 2007]. We show the normalized runtimes for the benchmarks in Figure 8. The columns to the left of the dotted line are the slowdowns when running on continuous power, with the left bar for an app denoting the unaltered code and the right bar the instrumented code. The apps with the largest slowdowns, `bmp-ext` and `cuckoo` had tasks with many frequently accessed local variables. Reserving registers for the runtime library parameters caused frequent spilling of the local variables to the stack, resulting in a large slowdown.

To see how the increase in run time effects intermittent execution, we also gather the runtimes of each app when running intermittently, shown on the right side of Figure 8. Each column denoting an app running on harvested energy is broken into two segments - on time, denoted with a purple color, when the device was running, and off time, denoted with grey, when the device was recharging. All segments are normalized to the base runtime of the unaltered app on continuous power. The grey segments that reach the top of the graph are cut off, with the true height noted in the number overlaying the bar. Power failures and recharge times are actual, not simulated, and are determined by the harvesting technology and physical environment. We can see that no matter whether instrumented or unaltered, the total time spent executing the app is dominated by recharge time. On average, each app spends 5.95 times as long charging as it does executing. Since the instrumented apps are much longer, they also take more energy to run, increasing the number of reboots necessary to run each app to completion. We show the number of reboots the instrumented and unaltered versions of the app took in Table 4. Note that the proportions of the instrumented reboot count to the unaltered reboot count are in line with the normalized slowdowns. The apps that rebooted the most, e.g., `bc` or `cuckoo`, have the largest difference between the continuous execution time and intermittent on time. This is because each reboot presupposes some amount of re-executed code, making the total number of instructions executed larger. While this increase in number of reboots and corresponding increase in runtime would definitely harm the performance of deployed code, it does not necessarily hamper the practicality of IBIS-D. An intermittence bug

can only be triggered if the device reboots, so causing more reboots on each run of an application can help to find bugs.

Table 4. Summary of reboot count of the unaltered and instrumented version of each app

Version	mpu^x	hdc^x	bmp^x	opt^x	tmp^x	wsn^x	elink^x	ar	bc	cem	cuckoo
Unaltered	6	4	2	3	5	4	5	3	4	3	4
Instrumented	37	32	106	23	61	23	17	136	299	193	323

We show a breakdown of the instrumentation calls executed in Table 5. Along with the number of calls to each instrumented function, we show the number of control dependence tainted variables, the number of data dependence tainted variables, and the number of variables cleared on task transitions. The last column in Table 5 shows whether IBIS-D detected the bugs reported by IBIS-S. We write "No bugs to confirm" for applications where IBIS-S did not report any bugs.

A key observation from the breakdown of instrumentation calls is that the full Alpaca applications did not contain any variables tainted by a control dependence (i.e., the **C. dep** column is 0). All the apps contained a task boundary before any control decisions off of I/O tainted variables since the apps do not require fresh data. This finding corroborates that IBIS-S is correct in reporting zero bugs across the Alpaca benchmarks as any RIO bugs will have control tainted variables, and is not missing any due to the imprecision in the static analysis.

All the true bugs detected by IBIS-S with task-filtering enabled were detected by IBIS-D except the one in elink, The tainted variables in elink were fields of a struct. Another field of the struct was involved in a WAR dependency, so Alpaca backed-up the entire struct, performing more conservatively than our software validation. IBIS-D correctly does not report a violation on variables that have both a WAR and RIO dependency, since the inconsistency will be fixed by the runtime. In the other applications, the bugs identified by IBIS-S did cause runtime memory violations that Alpaca did not fix, breaking the desired transaction semantics of the runtime, and showing that I/O idempotence violations are a real threat to correct intermittent applications.

Table 5. Runtime statistics of the apps

App	#Get	#Decide	#Control	#Set	#C. dep	# D. dep	#Cleared	Bugs Confirmed?
mpu^x	12149	7665	8873	1428	2448	1128	3060	Yes
hdc^x	8267	5453	4625	1005	0	1212	1608	No bugs to confirm
bmp^x	29043	13934	11229	2114	0	4536	3322	No bugs to confirm
opt^x	3785	2870	2345	408	714	516	1224	Yes
tmp^x	39805	19609	30501	7272	8282	7884	6060	Yes
wsn^x	7588	4115	5483	510	284	516	1375	Yes
elink^x	1845	2030	969	676	90	630	1324	Bug fixed by Alpaca
ar	53512	20604	35561	1512	0	2022	4032	No bugs to confirm
bc	113542	68304	65992	15	0	75	150	No bugs to confirm
cem	72110	55448	24143	128	0	390	768	No bugs to confirm
cuckoo	147753	105395	68197	1280	0	1286	2560	No bugs to confirm

8 CASE STUDIES

We found that I/O dependent idempotence bugs are most common in sensor drivers and low-level client applications–timing sensitive code that makes the bug difficult to fix with tasks or checkpoints. We present three bugs that IBIS found in figure 9, showing the simplified source code and bug-triggering execution traces. We discuss the consequences of the bugs and the challenges of fixing them while preserving timeliness and atomicity. The first two case studies come from the TI-RTOS `mpu` driver and the third from the `libwispbase` RFID decoder protocol.

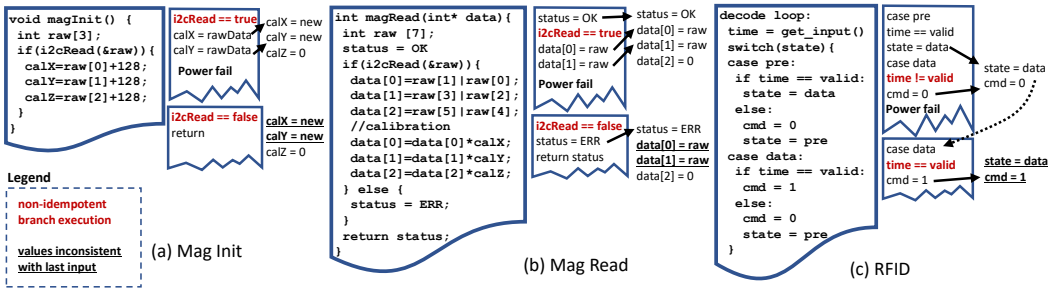


Fig. 9. Example bugs in real code. A solid arrow indicates a memory update caused by an instruction. A dashed arrow indicates a data dependency from NV memory to a re-executed instruction

The Magnetometer Initialization Function This function initializes calibration data used to scale reads from the magnetometer ((Figure 9(a)). The function checks that the sensor is powered on and then conditionally reads from the I2C bus. If the read was successful, the raw data read is stored into the X, Y, Z fields of the calibration structure. If the read was not successful, the function simply returns.

The bug occurs if the first read is successful, the calibration fields are partially updated before the power fails, and the second read is unsuccessful. On the first execution, the `calX` and `calY` fields are set to the most recent data. On the second execution, none of the fields are updated, and `calZ` remains inconsistent. The incorrect data will skew any future sensor reads, leading to incorrect behavior or crashes. The fix is simple; initializing the calibration fields before reading the new data will sanitize the variables on re-execution.

The Magnetometer Read Function This function reads data from the magnetometer (Figure 9 (b)). It performs checks to confirm that the magnetometer is ready, the buffer has not overflowed, and the I2C bus read returned correctly. If all the checks pass, the program transfers the magnetometer data stored in the temporary `raw` variable to the non-local data buffer, returning true. If a check fails, the program returns an error value and stores nothing into the non-local buffer.

The pattern is similar to the initialization bug described above: the bug manifests if the checks are passed and the data buffer is partially updated on the first execution, and on the second execution a check fails and the program returns. A key difference is that this function can return an error value. At first glance, this seems to render the data consistency issue moot: the programmer can know that the data read failed. Programmers may make the assumption, however, that the data buffer always contains the last successful sensor read, since the buffer is not touched if a check fails. Consequently, the programmer may use the data buffer even if the last read failed. Notice that this assumption is sound under continuous power and becomes invalid under intermittent power. To fix this bug, the programmer can initialize the data buffer before the branch.

RFID Protocol The RFID decoder protocol is a highly timing-sensitive state machine, as the state transitions to decode message preambles are based on the intervals between packet reception. If an entire sequence of messages fits the correct intervals, the protocol will enter the `data` state and interpret subsequent messages as the command. If any messages miss the correct interval, the machine resets to the starting state. In Figure 9 (c), we show a simplified version of the state machine with a preamble state and a data state.

The bug happens if a message with an invalid interval is received, and the program starts to reset the variables. Before the actual state variable is reset, the power fails. On reboot, some variables are consistent with the beginning of the protocol, such as the command variable, but the state machine itself never transitioned. The next message received has a valid interval, so the machine incorrectly

calls the `set_command`. This inconsistency can cause the machine to interpret a garbage sequence of messages as valid command.

The state variable exhibits both WAR and RIO dependencies, but critical timing dependencies in the protocol make it difficult to fix with tasks or checkpoints.

9 RELATED WORK

IBIS draws ideas from intermittent computing, program analysis (taint analysis in particular) and pattern-based bug finding and is related to memory consistency for non-volatile memory systems and fault tolerance in distributed systems. We discuss related work in each research area below.

Intermittent Computing Existing intermittent execution models are susceptible to I/O-dependent idempotence violations, requiring IBIS's support. Prior work collected automatic checkpoints [Balsamo et al. 2016, 2015; Jayakumar et al. 2014; Mirhoseini et al. 2013; Ransford et al. 2011], in some cases targeting idempotence violations stemming from WAR dependences [Bhatti and Mottola 2017; Hicks 2017; Van Der Woude and Hicks 2016]. DINO [Lucia and Ransford 2015], Chain [Colin and Lucia 2016] and Alpaca [Maeng et al. 2017] use tasks and buffer writes to ensure idempotent task re-execution, despite WAR dependences. RESTOP [Rodriguez Arreola et al. 2018], Sytare [Berthou et al. 2017], and Samoyed [Maeng and Lucia 2019] address retaining the state of peripheral devices. InK [Yildirim et al. 2018] and Coati [Ruppel and Lucia 2019] address event driven programming in intermittent systems. No prior system considers RIOs or addresses the input-dependent idempotence violations identified by IBIS.

IBIS provides support for validating bugs but not a debugging infrastructure, which is known to be hard in intermittent systems. Programmers can use existing debugging frameworks such as EDB [Colin et al. 2016] and Ekho [Zhang et al. 2011b].

Control-flow dependent bugs Non-idempotent control-flow dependent writes is a condition of the bugs that IBIS identifies. Several projects have identified control-flow dependencies as a key contributor to concurrency bugs or lack of bugs in distributed programs [Huang et al. 2014; Machado et al. 2016]. They are orthogonal as IBIS focuses on intermittent, not concurrent systems.

Timeliness and Atomic Events Mayfly [Hester et al. 2017] uses tasks and was the first paper to identify the timeliness problem for I/O in intermittent systems. Mayfly does not address the RIO problem, though it is not susceptible to them as tasks variables can only write to volatile memory. Capybara [Colin et al. 2018] is a HW/SW co-designed intermittent computing platform that enables atomic and reactive software events. Homerun [Kang et al. 2018] is an energy-harvesting system that supports atomicity for some I/O events. None of these systems considers the effect of RIO, making them ineffective for preventing input-dependent idempotence violations.

Persistent Memory Persistent memory enables recovery after power failures, but its presence introduces subtle software problems, especially in intermittent systems. Prior work studied how to keep persistent memory consistent for heap objects [Coburn et al. 2011; Volos et al. 2011] (including through the use of transactions), in-memory file systems [Dulloor et al. 2014], full system state [Moraru et al. 2013; Narayanan and Hodson 2012] and non-volatile intermittent processors [Ma et al. 2015a,b]. Prior modelling work [Pelley et al. 2014, 2015] defines persistency models that enable reasoning about non-volatile data consistency, but does not explicitly deal with intermittent execution nor I/O.

Taint Tracking, Program Analysis and Pattern-based Bug Finding IBIS-S uses static program analysis to find bugs because input-dependent idempotence violations match a clear, detectable pattern. Pattern based bug detection has seen success finding data-races [Savage et al. 1997], concurrency bugs [Lu et al. 2007a,b; Lucia and Ceze 2009; Lucia et al. 2010, 2011; Park et al. 2009, 2012, 2010; Shi et al. 2010], and performance bugs [Nistor et al. 2015, 2013]. Researchers have

applied static analysis to help find and fix concurrency bugs [Zhang et al. 2013] or to aid in enabling safe speculative parallelism, which considers similar correctness criteria to IBIS [Prabhu et al. 2010].

IBIS-S implements a coarse-grained static taint analysis to identify input-dependent branches. Taint analysis has been widely used to identify security-critical bugs cause by un-sanitized inputs, such as code injection attacks and format string vulnerabilities (e.g., [Jovanovic et al. 2010; Livshits and Lam 2005; Machiry et al. 2017; Ming et al. 2015; Yamaguchi et al. 2015]). IBIS-S' taint analysis trades precision for efficiency.

Dynamic taint tracking has been used to track information flow and security vulnerabilities at runtime [Chen and Kapravelos 2018; Enck et al. 2010; Lekies et al. 2013; Melicher et al. 2018]. IBIS-D is different from these systems in that it also tracks freshness.

Static analysis can also help validate intermittent systems via translation validation. Recent work presents an algorithm for verifying that the translation from a continuous program to a checkpointed intermittent system preserves program semantics [Dahiya and Bansal 2018]. This approach is orthogonal to IBIS and could be extended to prove the correctness of bug fixes of input-dependent idempotence violations in future work.

Idempotent Processing and Failure Atomicity Static analysis has been used to support idempotent processing [De Kruijf and Sankaralingam 2013; de Kruijf et al. 2012], which makes systems robust to failures. Idempotence analysis has also been used to make more efficient logging mechanisms for systems with persistent memory [Liu et al. 2018], and for implementing fault tolerance in distributed systems [Ramalingam and Vaswani 2013]. Dividing programs into idempotent regions precludes timeliness if inputs cannot be re-executed. JustDo logging [Izraelevitz et al. 2016] presents a mechanism for failure-atomic updates to persistent memory, but it also avoids re-executing code for better performance. Delay-Free concurrency [Ben-David et al. 2019] provides a construction for automatically transforming concurrent programs to be crash-consistent on persistent memory, but it focuses on shared data structures. Intermittent systems need the entire program to be checkpointed or in transactions, not just shared regions. IBIS has similar foundational ideas to these works on fault tolerance, e.g., how to resume the system from a consistent state and how to safely re-execute code manipulating persistent memory, but the resource and design constraints of distributed systems are quite different than that of embedded intermittent systems.

10 CONCLUSION

To reliably use energy harvesting devices as sensing platforms, programmers need to be able to reason about the effects of non-idempotent I/O, which is absent from prior work. This paper aims to detect bugs caused by repeated I/O operations in intermittent systems and presents IBIS, the first tool to characterize and detect bugs caused by non-idempotent I/O operations. IBIS finds instances of I/O-dependent, idempotence bugs primarily in sensor drivers and low-level applications. Since the timeliness constraints of OS level code make it difficult to use tasks or checkpoints, IBIS provides useful, necessary information for programmers to develop reliable intermittent programs.

ACKNOWLEDGEMENTS

This work was funded in part by NSF Career Award 1751029.

REFERENCES

Henko Aantjes, Amjad Y Majid, Przemyslaw Pawełczak, Jethro Tan, Aaron Parks, and Joshua R Smith. 2017. Fast downstream to many (computational) RFIDs. In *Proceedings of the 36th Annual IEEE International Conference on Computer Communications (INFOCOM '17)*.

Domenico Balsamo, Alex S Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M Al-Hashimi, Geoff V Merrett, and Luca Benini. 2016. Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 12 (2016), 1968–1980.

Domenico Balsamo, Alex S Weddell, Geoff V Merrett, Bashir M Al-Hashimi, Davide Brunelli, and Luca Benini. 2015. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters* 7, 1 (2015), 15–18.

Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. 2019. Delay-Free Concurrency on Faulty Persistent Memory. In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19)*. New York, NY, USA, 12.

Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. 2017. Peripheral state persistence for transiently-powered systems. In *Proceedings of the 2017 Global Internet of Things Summit (GIoTS '17)*. IEEE.

Naveed Anwar Bhatti and Luca Mottola. 2017. HarvOS: Efficient code instrumentation for transiently-powered embedded sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN '17)*.

Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering Information Leakage from Browser Extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 1687–1700.

Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*.

Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P. Sample. 2016. An Energy-interference-free Hardware-Software Debugger for Intermittent Energy-harvesting Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*.

Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '16)*.

Alexei Colin and Brandon Lucia. 2018. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction (CC '18)*.

Alexei Colin, Emily Ruppel, and Brandon Lucia. 2018. A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*.

Manjeet Dahiya and Sorav Bansal. 2018. Automatic Verification of Intermittent Systems. In *Verification, Model Checking, and Abstract Interpretation*, Isil Dillig and Jens Palsberg (Eds.). Cham.

Marc De Kruijf and Karthikeyan Sankaralingam. 2013. Idempotent code generation: Implementation, analysis, and evaluation. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '13)*.

Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static Analysis and Compiler Design for Idempotent Processing. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*.

Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*.

William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. Berkeley, CA, USA, 393–407.

EPCglobal Inc. 2015. EPC Radio-Frequency Identity Protocols Generation-2 UHF RFID. https://www.gs1.org/sites/default/files/docs/epc/Gen2_Protocol_Standard.pdf.

Xiaochen Guo, Engin Ipek, and Tolga Soyata. 2010. Resistive Computation: Avoiding the Power Wall with Low-leakage, STT-MRAM Based Computing. *SIGARCH Computer Architecture News* 38, 3 (June 2010), 371–382.

Josiah Hester and Jacob Sorber. 2017. Flicker: Rapid Prototyping for the Batteryless Internet-of-Things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (EnSys '17)*.

Josiah Hester, Kevin Storer, and Jacob Sorber. 2017. Timely Execution on Intermittently Powered Batteryless Sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (EnSys '17)*.

Matthew Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*.

Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. 337–348.

Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. New York, NY, USA, 16.

Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. 2014. QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *Proceedings of the 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*.

Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2010. Static Analysis for Detecting Taint-style Vulnerabilities in Web Applications. *J. Comput. Secur.* 18, 5 (Sept. 2010), 861–907.

Chih-Kai Kang, Chun-Han Lin, Pi-Cheng Hsiu, and Ming-Syan Chen. 2018. HomeRun: HW/SW Co-Design for Program Atomicity on Self-Powered Intermittent Systems. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '18)*. Article 29.

Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steven Glaser, and Martin Turon. 2007. Health Monitoring of Civil Infrastructures Using Wireless Sensor Networks. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN '07)*.

Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security (CCS '13)*. ACM, New York, NY, USA, 1193–1204.

Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. 2018. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 51)*. 258–270.

V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium (USENIX Security '05)*.

Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. 2007a. MUVI: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *Proceedings of the Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*.

Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2007b. AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants. *IEEE Micro* 27, 1 (Jan. 2007), 26–35.

Brandon Lucia and Luis Ceze. 2009. Finding Concurrency Bugs with Context-aware Communication Graphs. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*.

Brandon Lucia, Luis Ceze, and Karin Strauss. 2010. ColorSafe: Architectural Support for Debugging and Dynamically Avoiding Multi-variable Atomicity Violations. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*.

Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*.

Brandon Lucia, Benjamin P. Wood, and Luis Ceze. 2011. Isolating and Understanding Concurrency Errors Using Reconstructed Execution Fragments. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*.

Kaisheng Ma, Xueqing Li, Mahmut Taylan Kandemir, Jack Sampson, Vijaykrishnan Narayanan, Jinyang Li, Tongda Wu, Zhibo Wang, Yongpan Liu, and Yuan Xie. 2018. NEOFog: Nonvolatility-Exploiting Optimizations for Fog Computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*.

Kaisheng Ma, Xueqing Li, Shuangchen Li, Yongpan Liu, John Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. 2015a. Nonvolatile processor architecture exploration for energy-harvesting applications. *IEEE Micro* 35, 5 (2015), 32–40.

Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. 2015b. Architecture exploration for ambient energy harvesting nonvolatile processors. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA '15)*.

Nuno Machado, Brandon Lucia, and Luís Rodrigues. 2016. Production-guided Concurrency Debugging. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. 29:1–29:12.

Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security '17)*. Vancouver, BC.

Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 96 (Oct. 2017), 96:1–96:30 pages.

Kiwan Maeng and Brandon Lucia. 2019. Supporting Peripherals in Intermittent Systems with Just-In-Time Checkpoints. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*.

William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting. *2018 Network and Distributed System Security Symposium (NDSS)* (Feb 2018).

Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. 2015. TaintPipe: Pipelined Symbolic Taint Analysis. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15)*.

Azalia Mirhoseini, Ebrahim M Songhori, and Farinaz Koushanfar. 2013. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered ASICs. In *Proceedings of the 2013 IEEE International Conference on Pervasive Computing and Communications (PerCom '13)*.

Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. 2013. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems (TRIOS '13)*.

Dushyanth Narayanan and Orion Hodson. 2012. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*.

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100.

Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and Fixing Performance Problems That Have Non-intrusive Fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. Piscataway, NJ, USA.

Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. Piscataway, NJ, USA.

Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*.

Sangmin Park, Richard Vuduc, and Mary Harrold. 2012. UNICORN: A unified approach for localizing non-deadlock concurrency bugs. In *Software Testing, Verification and Reliability*, Vol. 25.

Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. 2010. Falcon: Fault Localization in Concurrent Programs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*.

Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceedings of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. Piscataway, NJ, USA.

Steven Pelley, Peter M Chen, and Thomas F Wenisch. 2015. Memory Persistency: Semantics for Byte-Addressable Nonvolatile Memory Technologies. *IEEE Micro* 35, 3 (2015), 125–131.

Prakash Prabhu, Ganesan Ramalingam, and Kapil Vaswani. 2010. Safe Programmable Speculative Parallelism. *ACM SIGPLAN Notices* 45, 50–61.

Proteus Digital Health. 2015. Proteus Digital Health. http://www.proteus.com/.

G. Ramalingam and Kapil Vaswani. 2013. Fault Tolerance via Idempotence. *Principles of Programming Languages (POPL)* (January 2013). https://www.microsoft.com/en-us/research/publication/fault-tolerance-via-idempotence/

Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*.

Alberto Rodriguez Arreola, Domenico Balsamo, Geoff V Merrett, and Alex S Weddell. 2018. RESTOP: Retaining External Peripheral State in Intermittently-Powered Sensor Systems. *Sensors* 18, 1 (2018), 172.

Emily Ruppel and Brandon Lucia. 2019. Transactional Concurrency for Intermittent Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*.

Alanson P Sample, Daniel J Yeager, Pauline S Powledge, Alexander V Mamishev, and Joshua R Smith. 2008. Design of an RFID-based battery-free programmable sensing platform. *IEEE Transactions on Instrumentation and Measurement* 57, 11 (2008).

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 391–411.

Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. 2010. Do I Use the Wrong Definition?: DeFuse: Definition-use Invariants for Detecting Concurrency and Sequential Bugs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*.

Jethro Tan, Przemysław Pawełczak, Aaron Parks, and Joshua R Smith. 2016. Wisent: Robust downstream communication and storage for computational RFIDs. In *Proceedings of the 35th Annual IEEE International Conference on Computer Communications (INFOCOM '16)*.

TI Inc. 2017a. MSP430FR59xx Mixed-Signal Microcontrollers (Rev. F). http://www.ti.com/lit/ds/symlink/msp430fr5969.pdf.

TI Inc. 2017b. TI-RTOS: Real-Time Operating System (RTOS) for Microcontrollers (MCU). http://www.ti.com/tool/ti-rtos-mcu Accessed: 2018-05-08.

Joel Van Der Woude and Matthew Hicks. 2016. Intermittent computation without hardware support or programmer intervention. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*.

Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*.

Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*.

Kasim Sinan Yildirim, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. 2018. InK: Reactive Kernel for Tiny Batteryless Sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems (SenSys '18)*. New York, NY, USA, 41–53.

Zac Manchester. 2015. KickSat. http://zacinaction.github.io/kicksat/.

Hong Zhang, Jeremy Gummeson, Benjamin Ransford, and Kevin Fu. 2011a. Moo: A batteryless computational RFID and sensing platform. *Department of Computer Science, University of Massachusetts Amherst., Tech. Rep* (2011).

Hong Zhang, Mastooreh Salajegheh, Kevin Fu, and Jacob Sorber. 2011b. Ekho: Bridging the Gap Between Simulation and Reality in Tiny Energy-harvesting Sensors. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems (HotPower '11)*. Article 9.

Wei Zhang, Marc de Kruijf, Ang Li, Shan Lu, and Karthikeyan Sankaralingam. 2013. ConAir: Featherweight Concurrency Bug Recovery via Single-threaded Idempotent Execution. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*.