Towards Instruction Level Record and Replay of Sensor Network Applications

Lipeng Wan
Department of Electrical Engineering
and Computer Science
University of Tennessee
Knoxville, Tennessee 37996
Email: lwan1@utk.edu

Qing Cao
Department of Electrical Engineering
and Computer Science
University of Tennessee
Knoxville, Tennessee 37996
Email: cao@utk.edu

***Abstract*—Debugging wireless sensor network (WSN) applications has been complicated for multiple reasons, among which the lack of visibility is one of the most challenging. To address this issue, in this paper, we present a systematic approach to record and replay WSN applications at the granularity of instructions. This approach differs from previous ones in that it is purely software based, therefore, no additional hardware component is needed. Our key idea is to combine the static, structural information of the assembly-level code with their dynamic, run-time traces as measured by timestamps and basic block counters, so that we can faithfully infer and replay the actual execution paths of applications at instruction level in a post-mortem manner. The evaluation results show that this approach is feasible despite of the resource constraints of sensor nodes. We also provide two case studies to demonstrate that our instruction level record-and-replay approach can be used to: (1) discover randomness of EEPROM writing time; (2) localize stack smashing bugs in sensor network applications.**

I. Introduction

One powerful technique for debugging software systems is the concept of *event-based record and replay*. Originally developed in the context of distributed applications [1], [2], [3], [4], this technique allows a developer to capture the execution-level events of software systems, especially those I/O and non-deterministic events, so that it can later replay the entire sequence of events precisely and deterministically. Because everything happens in exactly the same order as it happened during the original execution, a developer can observe any possible bugs in an *a posteriori* manner, so that the same problems or performance bottlenecks can be triggered and identified.

Until recently, this powerful technique has not been practically available for resource-constrained large-scale wireless sensor networks for multiple reasons. First, early software based approaches [5], [6], [7] have shown that replay-based methods directly borrowed from other domains of research tend to incur too much overhead to be useful for real-world applications. Later approaches, such as a recent solution in [8], focused on replaying only certain types of information, e.g., breakpoints in control flows. Although more practically feasible, such replays may not capture the real culprits of software bugs since a complete replay is beyond their capabilities. Second, hardware based approaches have been proposed recently as well [9]. However, these approaches require dedicated circuit boards to be built separately, which are usually tightly integrated with the particular microcontroller used by the sensor node. Although they can capture events at a high resolution, building such tools incurs considerable additional cost, and is usually not practical for average users.

The lack of effective tools contributes to one of the most challenging problems in sensor networks, namely, the lack of visibility at execution time. Therefore, debugging sensor network applications has been notoriously difficult. Furthermore, because of resource constraints, sensor nodes typically lack specialized memory protection mechanisms [10], [11]. Therefore, various bugs, such as race conditions and buffer overflows, can easily crash nodes and suspend normal network operations.

Our critical insight to address this problem is that due to the limited resources of individual nodes, we must synthesize both static (program level) and dynamic (run-time) information to achieve the goal of instruction-level replays. Specifically, we observe that, due to overhead, it is not realistic to collect fine-grained traces with a sufficiently high frequency that is comparable to conventional computer systems on embedded microcontrollers (e.g., at a rate of MHz) [1], [2], [3]. Moreover, since logging traces will change the timing of the original applications, their presence should be minimized to reduce the possibility of introducing additional bugs. Based on these concerns, we can only afford to collect relatively *coarse-grained* run-time traces for replay purposes. However, because such traces are so sparse, to make the later replays feasible, we must also exploit the static program information, and synthesize them together. In fact, the dynamic, run-time traces only serve as hints to complement with the static program information in our approach. Put in another way, our approach demonstrates that we can *infer* those remaining un-captured events through only those critical ones, while we prune unreached code blocks, impossible paths, and redundant traces, so that the total recording overhead can be minimized.

Based on this insight, we design and implement a novel software-based record-and-replay debugging technique for *a posteriori* replays of real executions of sensor network applications at the instruction level. We demonstrate that, by collecting run-time traces in a way that is guided by the analysis of the assembly code graphs, we can obtain entire replays of original applications down to the instruction level. The whole process is fully automatic and software-based, so that it can be readily adopted by application developers without the need for additional hardware.

The main contributions of our paper can be summarized in three aspects: first, we introduce the concept of the assembly level code graph model, called Assembly Code Graphs (ACGs), to represent the assembly code structure of sensor network applications. ACGs allow us to easily adopt well-known graph-based algorithms such as the Depth-First-Search (DFS) technique to find all possible execution paths. Second, we develop a run-time trace logging approach to record critical events and timing information during program execution. We then develop an algorithm to synthesize such recorded information with the analysis results of the static ACGs to obtain the actual execution paths. Finally, we perform extensive experiments with six benchmark applications to evaluate the correctness and overhead of our approach. We also provide two case studies to demonstrate how to use our instruction level record-and-replay approach to: (1) discover randomness of EEPROM writing time; (2) localize stack smashing bugs in sensor network applications.

We also clarify several issues that are *not* handled by our approach. First, since this approach requires recording program execution traces, it may not be able to handle potential security attacks, where the attackers may easily disable the trace recording functionality. Indeed, while we believe our proposed approach can be highly useful for detecting bugs that are caused by programmers' mistakes, it may not be able to defend against carefully crafted malicious attacks. Second, this approach does not assume the use of extra hardware. Therefore, it does not record all sources of non-determinism. In fact, we argue that in embedded sensor networks, recording all non-deterministic events, such as all sensor readings and hardware variations, is infeasible given the limited storage space. Indeed, our approach is based on its ability to *infer* actual paths with *incomplete* information. Finally, though very rare in typical sensor network applications, some carefully crafted code may need additional run-time information to be collected for our inference algorithm to work properly. We will discuss this scenario in Section VII.

The remaining of the paper is organized as follows. In Section II, we present the idea of Assembly Code Graph model and its associated operations. In Section III, we introduce an automatic instrumentation technique to collect run-time traces, as well as the synthesizing algorithm to prune unreached code blocks or execution paths. We evaluate our record-and-replay system and analyze the evaluation results in Section IV. To demonstrate the effectiveness of our approach, we apply it to discovering non-determinism of EEPROM writing and detecting stack smashing bugs of sensor network applications in Section V. Section VI provides a survey of several existing works closely related to our approach, and illustrates their differences. Finally, we make concluding remarks and discuss future work in Section VII.

II. Assembly Code Graph

In this section, we describe the model of assembly code graphs (ACGs), and their associated operations. During the following description, we assume the target platform to be AVR-based microcontrollers such as those used in MicaZ and IRIS motes. The approaches can also apply to other similar platforms, such as MSP430, as long as the source code is written in the C programming language (or, as is the case with NesC used by TinyOS, can be converted to the C programming language).

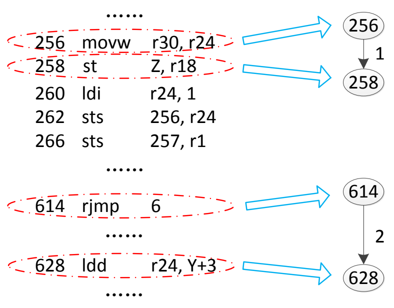
A. Graph Structure of Assembly Code

We use assembly code of an application in the construction of ACGs, though the same technique also applies to binary instructions. Formally, these instructions are classified into five categories [12]: arithmetic and logic instructions, branch instructions, data transfer instructions, bit and bit-test instructions, and MCU control instructions. Among them, only branch instructions, such as JMP, CALL, RET, can change the execution path of a program. In other words, if a sequence of assembly code does not contain any branch instructions, these instructions should be executed sequentially.

Motivated by this observation, we model the structure of assembly code in the form of a directed graph. Specifically, we use $G(V, E)$ to represent a directed graph of assembly code. Here, V is the set of vertices, and E is the set of directed edges. Each vertex $v \in V$ represents one instruction read from the ELF file (ELF file contains the assembly instructions compiled from C source code by the standard AVR-gcc compiler). For two vertices v_1 and v_2, if v_2 is the very next instruction following v_1 in an execution sequence, there is a directed edge $e \in E$ from v_1 to v_2. We also define the cost of e to be the number of CPU cycles consumed by instruction v_1.

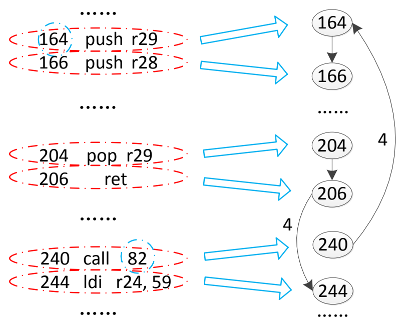
B. Construction of Assembly Code Graph

The key step to construct an ACG is to find all possible direct successors of each vertex. For an instruction that is not a branch instruction, this is simple, as its direct successor must be the next instruction following it. For example, as shown in Fig. 1, instruction "`ld r24, Z`" is the next instruction after "`movw r30, r24`", so there is a directed edge from "`movw r30, r24`" to "`ld r24, Z`". Similarly, for a branch instruction, we decide the direct successor by finding out where this branch leads to using its operand as the offset in the address. For example, if the branch instruction is "`rjmp 6`", and suppose the address of this branch instruction in memory is $addr_{rjmp}$, then the address of its direct successor should be $addr_{rjmp}+(6+1)\times 2$ (here, in accordance with AVR assembly conventions, we assume that the instruction "rjump 6" specifies the offset in words, and each word is two bytes, and the "+1" is due to that the offset is usually set according to the address of the following instruction, not the calling instruction itself), which is also illustrated in Fig. 1. However, not every branch



```
......
256  movw   r30, r24
258  st     Z, r18
260  ldi    r24, 1
262  sts    256, r24
266  sts    257, r1
......
614  rjmp   6
......
628  ldd    r24, Y+3
......
```

Fig. 1. A Simple Example for ACG Construction



```
......
164  push r29
166  push r28
......
204  pop  r29
206  ret
......
240  call 82
244  ldi  r24, 59
......
```

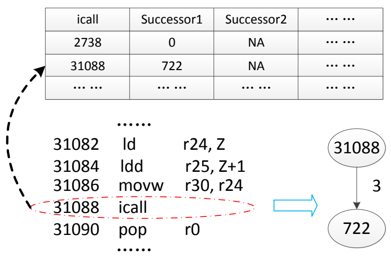
Fig. 2. Find the Corresponding “ret” for a “call”

instruction’s successor can be found so easily. In fact, finding the successors of some branch instructions, such as “ret”, is more complicated.

To demonstrate how to find successors of “`ret`” instruction, we use the following example as shown in Fig. 2. When instruction 240 “`call 82`” is read, based on its operand “82”, we can obtain the address of the starting instruction of the function being called as $82 \times 2 = 164$, and add a directed edge between instruction 240 “`call 82`” and instruction 164 “`push r29`”. Then we need to find “`ret`” instruction for this “`call`” instruction, and the searching starts from instruction 164 “`push r29`”. During the searching, we may encounter another “`call`” instruction, which will lead to another nested “`call-ret`” sequence. In this case, as the “`ret`” will continue with the next instruction, the searching will proceed to the next instruction that is executed after this “`call`” returns. As shown in Fig. 2, when the instruction 206 “`ret`” is read, a directed edge will be added between instruction 206 “`ret`” and instruction 244 “`ldi r24, 59`”, which is the very next instruction of instruction 240 “`call 82`” in memory [1]

Besides the “`call`” and “`ret`” instructions, there is another instruction named “`icall`” that deserves special handling. This instruction represents an indirect call to register Z. Based on static analysis, we cannot know what is the next instruction that will be executed, because its next instruction depends on the value of Z register at run-time. In such cases, we

[1]Note that the GCC compiler may sometimes generate sequences of instructions using `call` instructions without corresponding `ret` instructions. Our searching strategy will take care of such exceptions by not requiring backward directed edges to be inserted.



icall	Successor1	Successor2	... ...
2738	0	NA	... ...
31088	722	NA	... ...
... ...	... ...	... ...	... ...

```
......
31082  ld     r24, Z
31084  ldd    r25, Z+1
31086  movw   r30, r24
31088  icall
31090  pop    r0
......
```

Fig. 3. Addressing the Special Case of the “`icall`” Instruction

could not simply record the value of the Z register at run-time as it will cause too much overhead. Instead, we carry out a per-application pre-profiling strategy that allows us to run each application and record all the possible destination addresses of “`icall`” instructions into a table. Then during the construction of an ACG, when an “`icall`” instruction is encountered, we fetch its next instruction from the table directly without having to know the value of Z register. This process is fully automatic, and only needs to be executed once for each compiled application. Since the use of “`icall`” is rare and exhibits fixed patterns, this preprocessing is highly lightweight. An example on the way to handle the “`icall`” instruction is shown in Fig. 3, where the instruction 31088 is an “`icall`” instruction, which has only one successor, instruction 722, based on the lookup table. The ACG construction algorithm will add an directed edge between instruction 31088 and 722, and the associated “`ret`” instruction of this “`icall`” will also be handled like we mentioned before.

C. *Inference of Possible Execution Paths*

Observe that for each execution, a program’s instructions form a unique path on the Assembly Code Graph. However, only based on the Assembly Code Graph or any other static information of the source code, we cannot achieve the goal of execution trace reconstruction, because there might be different paths between two nodes on the graph, and the path may also contain an uncertain number of loops. If we want to identify the execution path uniquely, we must know which path the program has chosen and how many times each loop on the path has been executed. All such information can only be obtained during run-time. Therefore, we need to record additional run-time traces of the program. We describe our approach in two steps. First, in this section, we discuss how to find all possible paths between two nodes. In the next section, we discuss how to collect additional run-time traces, which serve as shepherds to guide a unique reconstruction of the actual execution path.

The first problem is formulated as follows: given a starting node and an ending node on the Assembly Code Graph, find all possible paths between these two nodes on the graph and all possible loops on these paths. Our approach to solve this problem adapts the DFS (Depth-First-Search) algorithm in graph theory, where we report the loops whenever they are discovered, but do not traverse the loops so that the algorithm will terminate in finite amount of time. Detailed steps are

Algorithm 1 Find Possible Execution Paths

```
Input: ACG $g$, Instruction vertex $v$ and $v_{end}$.
Output: Possible execution paths and loop paths.
  if $v$ is the ending vertex then
    Mark $v$ as visited;
    Add $v$ to $LoopDetcting$ and $PossiblePathFinding$;
    Back up all LIFO queues;
    findPossiblePath($g, v_{next}, v_{end}$);
    Recover all LIFO queues;
    Store $PossiblePathFinding$ into file;
    return
  else if $v$ has been visited then
    Search $v$ in $LoopDetcting$;
    Store vertices from $v$ to the end of $LoopDetcting$;
    Add $v$ to $LoopDetcting$ and $PossiblePathFinding$;
    return
  else
    Mark $v$ as visited;
    Add $v$ to $LoopDetcting$ and $PossiblePathFinding$;
    if $v$ is a "call" instruction then
      Add $v$ to $CallStack$;
    end if
    if $v$ is a "ret" instruction then
      Pop out the "call" on top of $CallStack$;
      Mark vertices from the "call" to "ret" as unvisited;
      Back up all LIFO queues;
      findPossiblePath($g, v_{retnext}, v_{end}$);
      Recover all LIFO queues;
    else
      for all $v_{next}$ do
        Back up all LIFO queues;
        findPossiblePath($g, v_{next}, v_{end}$);
        Recover all LIFO queues;
      end for
    end if
  end if
```

presented in Algorithm 1. Specifically, it implements the recursive function findPossiblePath(), which has three input parameters: g represents the ACG of the program, while v and v_{end} represent the starting and ending nodes, respectively. In this algorithm, we maintain three LIFO (last in, first out) queues: `PossiblePathFinding`, `LoopDetecting`, and `CallStack`. `PossiblePathFinding` is used to record the vertices that might be on a possible path without loops, `LoopDetecting` is used to record the vertices that might be on a possible loop, and finally, `CallStack` is used to record the "`call`" instructions that have been met during the searching. There are two other notations in Algorithm 1: $v_{retnext}$ represents the successor instruction of the "`ret`" instruction, and v_{next} represents the successor instruction of v. The remaining of the algorithm is similar to DFS, and therefore, is not elaborated.

III. Logging and Replaying with Runtime Traces

Given a list of all possible paths generated by the algorithm we discussed earlier, our next task is to identify the *real* path through analysis of run-time traces. This task is feasible based on two observations: first, the number of CPU cycles consumed by each instruction is known in advance; second, we can selectively record certain run-time traces to remove those uncertainties introduced by branches, loops, and interrupts. We next describe our approach in detail.

A. Timing-aware Logging of Critical Events

To reconstruct the execution path, we need to selectively record run-time traces. To minimize system overhead, we must introduce as little logging overhead as possible. Specifically, we use an approach similar to the one used in PowerTOSSIM [13] to obtain how many times each basic block (a basic block is a portion of code with no branches) has been executed. The key idea is to instrument the source code with the C Intermediate Language (CIL) library [14], during which a snippet of logging code is automatically inserted into each basic block. In the logging code, we collect the following types of information for each basic block:

- The latest timestamp this basic block was executed.
- The number of times this basic block was executed.

Note that to reduce overhead, we do not record the timestamp whenever a basic block is executed. Instead, only the latest execution is recorded, as we find this saves the storage space considerably. Furthermore, if there is only one loop on the execution path and if the time interval between the starting point and the ending point of this execution path is known, we can easily derive how many times this loop has been executed. On the other hand, if we need to replay the execution of nested loops, this method may not be enough and requires additional information. We will discuss this case later through an example.

Besides control blocks, we also obtain traces on the return addresses of interrupts. In most cases, we cannot know when and where an interrupt will occur before the program is executed. Without such information, we are not able to replay the actual execution based on the constructed ACG. Therefore, we need to record the return addresses of the interrupts. We achieve this goal by reading the value of stack pointer register in the interrupt handler, from which we derive the value of the return address. Since the number of interrupt handlers for an application is a fixed, limited value, we can afford to add a few lines of code to record the return addresses of these interrupt handlers without incurring excessive overhead. Finally, when the original program execution is within a loop, we also store the current timestamp of the interrupts, so that we can infer the number of loops that have been executed when the interrupt occurs.

B. Replaying Execution Traces

We now describe the steps to replay execution traces collected from one run of the original application. This process is straightforward for those applications that only have one loop or multiple disjoint loops. Since the disjoint loops must be executed one after another, given that we have recorded the execution counter for each basic block, we can easily derive how many times a loop is executed by examining the value of execution counters of basic blocks in this loop. The following is an example that shows the process of finding the execution path with ACG and run-time traces.

TABLE I
PROCESSED RUN-TIME TRACE DATA I

Basic Block ID	# of Times Executed	# of CPU Cycles Consumed	Instruction ID
0	1	0	212
1	0	0	430
2	0	0	516
3	0	0	594
4	1	88	778
5	1	136	836
6	977	49959	894
7	0	0	970
8	1	0	1074
9	1	24	1142

We use the Blink program as an example, which has been instrumented by the CIL library under Avrora [15]. It generates run-time traces as shown in TABLE I. We then choose the instruction 838 "`ldd r25, Z+1`" (next instruction of instruction 836) as the starting node, and the instruction 894 "`ld r24, Z`" as the ending node. Then we apply Algorithm 1 to find a total of four possible execution paths and two loops between these two nodes. By calculating the number of CPU cycles consumed by each possible execution path and loop path, we find that only one execution path is consistent with the recorded run-time traces, which consumes a total of $47 + 51 \times (977 - 1) = 49823$ CPU cycles, the same as the $49959 - 136 = 49823$ CPU cycles between basic block 5 and basic block 6 in TABLE I).

Although the example above is relatively straightforward, a more challenging scenario is when a loop contains another loop, as illustrated in a revised Blink application shown in Fig. 4 (loop 1 contains not only basic blocks 5 and 7, but also loop 2, which consists of basic block 6). To replay the actual execution in these two loops, we need to infer how many times loop 2 has been executed in each instance of loop 1. In this case, the run-time trace is not enough because it only provides summarized information such as how many times loop 1 and loop 2 have been executed at the end. Therefore, we adopt an alternative strategy where we resort to the source code of the programs to statically find the number of times a loop will be executed. In WSN applications, this can usually be inferred as they are typically constants. If non-constant number of loops is encountered, our analysis method will require the user to re-execute the application to dynamically gather more information, including the latest timestamps of multiple rounds of execution of each basic block, e.g., the latest timestamp of the basic block 6 will be recorded multiple times. By collecting such additional information as hints, we can infer the actual path execution successfully during the remaining inference procedure.

Finally, we describe an ACG transformation approach to address the scenario when an interrupt occurs. In this case, the program will execute the interrupt handler immediately after the current instruction. As the normal execution flow has been changed, the constructed ACG needs to be modified temporarily, so that the inferred execution path reflects the presence of the interrupt handler. We perform the following steps for this purpose: first, we remove the directed edge from

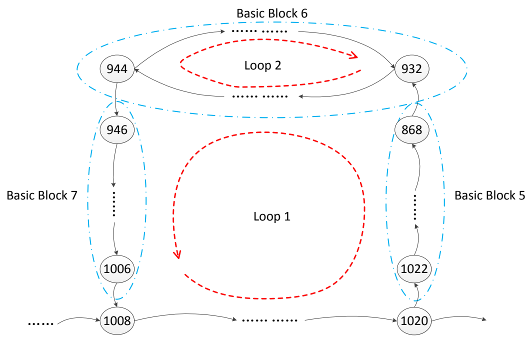


Fig. 4. A Loop Contains Another Loop

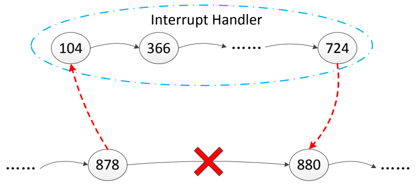


Fig. 5. Modified Assembly Code Graph

the current instruction u to its immediate successor v. Second, we add a directed edge from u to the first instruction w of the interrupt. Finally, we add a third edge from the "`reti`" instruction of the interrupt handler to v. Note that we can obtain the addresses of u and v by inspecting the return address of the interrupt handler, which is always kept with a fixed offset in the stack. As an example, Fig. 5 illustrates the scenario where an interrupt handler is added to the ACG between instruction 878 and 880. After finding the execution path that covers the interrupt handler, we need to change the modified ACG back to the previous state since the program will continue to run on the previous ACG after the interrupt finishes.

C. *Implementation Details*

In this section, we describe the implementation details of our logging and replaying system. Our implementation is customized based on the properties of the underlying hardware platform, i.e., the MicaZ nodes. It contains two components: code instrumentation and trace collection.

In the code instrumentation component, we exploit the CIL framework to transform the original C code into instrumented code. Specifically, the CIL framework will automatically insert variables into each basic block of the original code, so that it can record how many times each of these basic blocks are executed. In addition, we exploit the cycle-accurate timers on the AVR platform to maintain timing information, which allows timestamps to be recorded when these basic blocks are reached. For each basic block, its associated counter will be increased and its corresponding timestamps will be updated whenever it is executed. In practice, we find that this approach leads to highly compact logging traces.

In the trace collection component, we use the on-board 512KB flash to store collected traces. Specifically, all traces are stored in separate files in a simple file system, which allows easy transfer at the end of the experiment. Once the experiment finishes, we copy these files through the wireless radio for data analysis. Because such files are stored in a very compact form, a specialized parser is implemented to parse the file contents and reconstruct the original traces.

IV. Evaluation

In this section, we evaluate the performance of the record-and-replay approach to show that with the run-time traces, it can replay the execution of a WSN application accurately and effectively at the assembly code level.

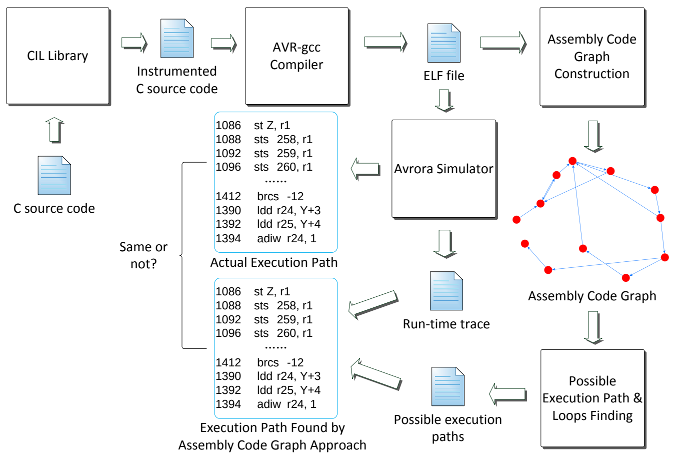
A. Evaluation Design

The most challenging part for evaluating our approach is that, ideally, we need to find the actual execution path of the assembly instructions as the ground truth, so that we can compare them with the inferred sequences. However, in practice, such ground truth data are not possible to obtain with existing hardware. Given that it is much easier to get the ground-truth execution path if the code was executed on a simulator, in our evaluation, we turn to an alternative, simulator-based approach. The intuition is simple: if the replay results can be shown to be accurate for the programs executed by a high-fidelity, cycle-accurate simulator, then we are relatively confident to claim that this approach should also be able to accurately replay the execution of programs running on real hardware.

After comparing the pros and cons of several simulators, we choose Avrora, a set of simulation and analysis tools for programs written for the AVR microcontroller. In our experiments, we modified Avrora according to our needs, and let it print out assembly instruction sequences during the execution of any simulated applications.

The entire procedure of the evaluation is illustrated in Fig. 6. Given a sensor network application written in C, we use the CIL library to instrument the original source code so that a logging segment is inserted in each basic block. Next, we run this application under Avrora, and record the run-time traces generated by these logging segments. At the same time, we also construct the ACG of the instrumented code, and find all possible execution paths. Based on the run-time traces, we infer the real execution path. Finally, we compare this inferred execution path with the *actual path* produced by Avrora during run-time to evaluate the accuracy of the replay.

Here one may come up with this question: now that the sensor mote simulator like Avrora can help us understand the execution of the WSN programs, why don't we just use Avrora to replay the execution in the first place? The reason is that simulators such as Avrora will typically emulate physical devices, e.g., MicaZ nodes, in a deterministic way. Therefore, the simulation results of a WSN program will always be the same no matter how many times it is executed. For example, although on real sensor motes, the time to perform an EEPROM write operation is considerably random due to hardware variations (we will discuss this example in Section V), Avrora will always adopt a constant, hard-coded value for EEPROM writing time. In other words, the advantage of



```
1086   st Z, r1
1088   sts  258, r1
1092   sts  259, r1
1096   sts  260, r1
         ......
1412    brcs  -12
1390    ldd r24, Y+3
1392    ldd r25, Y+4
1394    adiw  r24, 1
```

```
1086   st Z, r1
1088   sts  258, r1
1092   sts  259, r1
1096   sts  260, r1
         ......
1412    brcs  -12
1390    ldd r24, Y+3
1392    ldd r25, Y+4
1394    adiw  r24, 1
```

Fig. 6. Procedure of Evaluation

TABLE II
WSN Applications Used for Evaluation

Application	SLOC (C)	SLOC (Assembly)	Functionality
Blinking	112	446	Toggle the LED
Sensing	163	641	Read the light sensor
Blinking2	1600	4525	Toggle the LED application (under TinyOS)
Broadcasting	6323	10486	Broadcast packets (under LiteOS)
Receiving	6278	10207	Receive packets (under LiteOS)
Forwarding	6759	10534	Receive packets, then forward them to next hop (under LiteOS)

our approach over Avrora is that it captures all uncertainties through run-time traces, so that the results of the replay will faithfully reflect the actual execution of WSN programs.

B. Experimental Results

1) Accuracy of Replay: To evaluate the accuracy of the replay approach, we choose six different WSN applications as shown in Table II. The first two applications are written based on AVR LibC library [16] without any operating system support while the latter four are implemented based on existing operating systems for sensor networks such as TinyOS [17]. The functionality of these applications are also described in Table II. The term SLOC refers to the number of lines of source code, which is often used to illustrate the size of the source code of the program. The AVR microcontroller runs at $7.3728MHz$. We execute the applications on the Avrora simulator for comparison purpose.

We define the accuracy of the replay as the ratio of the number of instructions that are replayed accurately to the number of instructions that were actually executed by the simulator. Table III shows the replay results, where we can observe that for these six application benchmarks, all executed instructions are replayed correctly (we didn't take the instructions executed before the timer/counter started into account since they are just a small number of initialization instructions generated by the compiler).

TABLE III
REPLAY RESULTS OF DIFFERENT WSN APPLICATIONS

Application Name	# of Instructions Executed	# of Instructions Replayed	Accuracy (%)
Blinking	20239596	20239596	100
Sensing	39229424	39229424	100
Blinking2	2554101	2554101	100
Broadcasting	902564	902564	100
Receiving	837016	837016	100
Forwarding	1027501	1027501	100

TABLE IV
CODE OVERHEAD OF TRACE LOGGING

Application Name	Lines of Original Code (C)	# of Basic Blocks	Lines of Inserted Code (C)
Blinking	82	10	30
Sensing	127	12	36
Blinking2	1219	127	381
Broadcasting	4859	488	1464
Receiving	4829	483	1449
Forwarding	5271	496	1488

TABLE V
STORAGE OVERHEAD OF TRACE LOGGING

Application Name	Storage Overhead for Basic Blocks	Storage Overhead for Interrupts
Blinking	60B	0.88KB/s
Sensing	72B	0.88KB/s
Blinking2	762B	0.88KB/s
Broadcasting	2928B	0.88KB/s
Receiving	2898B	0.88KB/s
Forwarding	2976B	0.88KB/s

2) Overhead of Trace Logging: We now evaluate the overhead brought by the additional logging code inserted into the applications. We first measure how many lines of extra code have been inserted into the original code. Since we just insert three lines of C code in each basic block, we can calculate the total number of lines of extra code based on the number of basic blocks which are shown in Table IV.

We also measure the storage overhead of the logging process. This overhead comes from two sources, the first one is the global array for storing the execution counter and latest timestamp of each basic block, and the second is the global array for storing the return addresses and timing information for each interrupt handler. While the first overhead is fixed, the overhead for interrupts is linearly proportional to the frequency of interrupts. In all six benchmark applications, the timer interrupt occurs every $50,000$ CPU cycles for the need to calculate timestamps. This leads to an overhead about $0.88K$ bytes per second. Therefore, the installed flash of MicaZ nodes can support experiments up to 10 minutes in lab environments for these applications. In real deployments, additional flash memory can be installed to expand the time length for storing event data [18]. Table V shows the overall overhead results.

3) Energy Consumption: To compare the energy consumption of each of our benchmark applications before and after code instrumentation, we run the original code and the instrumented code of the above 6 benchmark applications on Avrora for five minutes, respectively. The goal of this experiment is that, by exploiting the energy monitor module of Avrora, we can obtain the estimated energy consumption of each hardware component at the end of the execution, such as CPU, flash, radio, among others. The results, which are measured in Joules, are shown in Fig. 7.

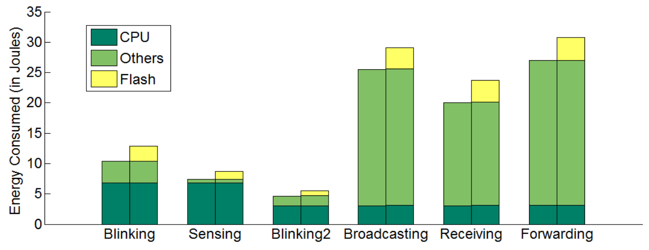


Fig. 7. Energy Consumption of Benchmark Applications

In this figure, observe that for each benchmark application, the left bar represents the energy consumption of the program without code instrumentation, while the right one represents the energy consumption of the program with code instrumentation. The energy consumed by each program is divided into three parts, including the energy consumed by CPU, by the flash, and by other hardware components (such as LEDs, radio, etc). We can also observe that the code instrumentation almost does not increase the energy consumption of hardware components except for flash. The later is simply because for the instrumented program, we need to store the run-time trace data into flash for later retrieval.

V. CASE STUDIES

A. Revealing Randomness in EEPROM Writing Time

According to the data sheet of ATmega128 microcontroller [19], the typical programming time for EEPROM access from the CPU is $8.5ms$ (8448 calibrated RC oscillator cycles[2]), which seems quite accurate and certain. However, when we analyze the run-time traces collected from the real sensor motes equipped with ATmega128 microcontroller, we find that the time spent on writing multiple bytes to EEPROM is a random variable rather than a fixed value. Actually, even we run the same program on the same sensor mote several times, each time the EEPROM writing time will be different. Such randomness might cause some "subtle" bugs which are difficult to detect.

When programming sensor network applications, developers often are not concerned on how their code works at the instruction level, or even the very existence of non-determinism during EEPROM write operation. Even the newest control flow tracing techniques such as TinyTracer cannot handle such non-determinism properly as their code instrumentation was done in source code level. In other words, they cannot insert any logging code into the assembly instructions (as shown in Fig. 8) generated by AVR-gcc compiler, which makes it impossible to capture such non-deterministic operations.

In our approach, with the introduction of ACGs, we can accurately capture and infer such non-deterministic sources, so that full application replays are made possible. In this case study, we demonstrate this point in detail. As shown in Fig. 8, during the operation of writing 16 bytes into

[2]Configured with `1MHz` clock, independent of CKSEL-fuse settings

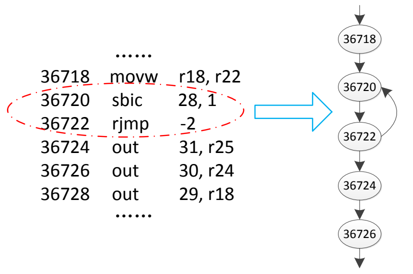


Fig. 8. Assembly Code for Writing EEPROM Generated by AVR-gcc

TABLE VI
RANDOMNESS IN EEPROM WRITING TIME

Measurement	CPU Cycles Consumed	# of Loops
1	919953	306526
2	919974	306533
3	920022	306549
4	919941	306522
5	919869	306498
6	919902	306509

EEPROM, the program will keep running in the loop between instruction 36720 and 36722. The number of times this loop will be executed is affected by hardware variations. During the construction of ACG, we detect this loop, and insert logging points before and after it. By subtracting the timestamps of these two logging points, we can derive how many times this loop has been executed. The results are shown in Table VI (Here the CPU frequency was $7.3728 MHz$).

B. Detecting Smashed Stacks in Sensor Network Applications

In this section, we describe how to exploit our record-and-replay system to detect live bugs in real applications. We focus on stack smashing (also known as stack buffer overflow) bugs, which occur when a program writes to a memory address on the program's call stack outside the intended area. Such bugs are particularly devastating to sensor nodes, since they do not have sophisticated memory protection mechanisms. Our following case study demonstrates that instruction level record and replay is particularly effective in revealing vulnerabilities caused by stack smash bugs. The unsafe code we choose belongs to the CC2420 radio driver of the MicaZ hardware platform, as shown below.

```
inline void cc2420radiom_PacketRcvd(void) {
  Radio_MsgPtr pBuf;
  uint8_t tmp_buff[4];
  int i;
  ...
  pBuf = cc2420radiom_Receive_receive((Radio_MsgPtr)
       pBuf);
  i = 0;
  //unsafe code starts here
  while(i < pBuf->length) {
    //vulnerability
    tmp_buff[i] = pBuf->data[i];
    i++;
  }
  ...
  buffer = tmp_buff;
}
```

TABLE VII
THE LAYOUT OF STACK

Memory Address	Usage	Value before Buffer Overflow	Value after Buffer Overflow
0x10ff	End of IRAM	...	...
...	...	...	...
0x10ea	other	...	0x00
0x10e9	other	...	0xdb
0x10e8	$@ret_H$	0x1e	0x00
0x10e7	$@ret_L$	0x7f	0xdb
...	...	...	...
0x10df	other	...	0xdb
0x10de	tmp_buff[3]	0x00	0x00
0x10dd	tmp_buff[2]	0x00	0xdb
0x10dc	tmp_buff[1]	0x00	0x00
0x10db	tmp_buff[0]	0x00	0xdb

Note that here, we insert some unsafe code similar to those used in [20] into a large application such that under certain circumstances, a stack buffer overflow will be triggered. In this program, **tmp_buff** is a local array with a size of 4 bytes, and **pBuf** is a pointer pointing to the received packet. The **while** loop copies **pBuf->length** bytes of data from the payload of received packet into array **tmp_buff**. If the payload size of the received packet (**pBuf->length**) is set to a value larger than 4, a stack buffer overflow will occur.

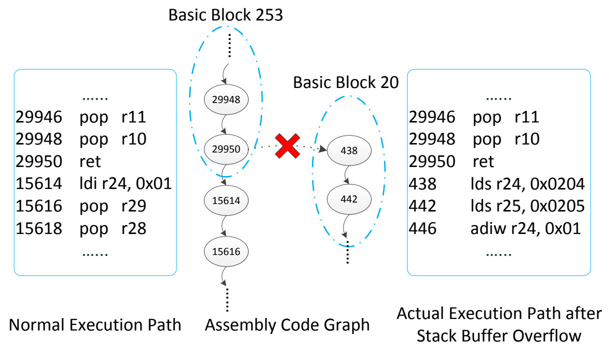
Now we construct a mal-packet which has a payload of size 16 bytes shown as follows.

```
uint8_t payload[16] = {
  0x00, 0xdb, 0x00, 0xdb,
  0x00, 0xdb, 0x00, 0xdb,
  0x00, 0xdb, 0x00, 0xdb,
  0x00, 0xdb, 0x00, 0xdb
};
```

Once the sensor node that contains the unsafe code introduced above receives this packet, a buffer overflow will occur and the stack of the sensor node will be smashed. The layout of the stack before and after the buffer overflow is shown in Table VII.

According to this table, we know that the high 8-bit and low 8-bit of the return address ($@\texttt{ret}_H$, $@\texttt{ret}_L$) of this packet receiving function are stored on the stack at address **0x10e8** and address **0x10e7**, respectively, and the normal value of the return address before the buffer overflow is **0x1e7f**. However, because of the lack of the bound checks, more data were written to the buffer **tmp_buff** than it could store, and the new data overlapped with the original data stored on the stack, including the return address of this function. As a result, when this function returns, it will return to **0xdb** instead of **0x1e7f**.

We now demonstrate how our ACG based instruction level record-and-replay approach can detect this stack buffer overflow bug. First, we instrument the C code of the sensor application to construct its ACG. Second, we run this unsafe sensor application to receive packets that contain relatively long payloads that will trigger this bug. After collecting the run-time traces from the node on which stack buffer overflows occurred, we feed the traces to our replay approach to find the execution path on the ACG of the unsafe program. During its execution, however, since the return address of the function has been changed from **0x1e7f** (instruction 15614) to **0xdb**



```
......
29946   pop  r11
29948   pop  r10
29950   ret
15614   ldi r24, 0x01
15616   pop  r29
15618   pop  r28
......
```

```
......
29946   pop  r11
29948   pop  r10
29950   ret
438     lds r24, 0x0204
442     lds r25, 0x0205
446     adiw r24, 0x01
......
```

Fig. 9. Stack Buffer Overflow Detection

(instruction 438) as the result of the stack smashing, the actual execution path between basic block 253 and basic block 20 does not exist on the normal ACG, as shown in Fig. 9. If the execution path could not be found on the ACG, our replay approach will automatically generate an exception and report the ID of the last basic block appeared on the replay path, so that the developer can localize the potential stack buffer overflow bug.

Note that, in the above case, we focus on how to use our record-and-replay approach to detect stack smashing bugs thereby keeping sensor network applications from failures. Such bugs are usually caused by programmers' unintentional mistakes. On the other hand, however, if such stack smashing bugs are used to launch a code injection attack, our approach probably cannot detect the source as the malicious code would probably choose not to enable any instrumentation. We emphasize that the latter case will require other techniques to handle, and is out of this paper's scope.

VI. Related Work

With extensive use of wireless sensor networks in different areas, the debugging techniques for WSN applications have become more and more critical. Many existing works can be classified under this topic, such as [21], [22], [23], [24]. In this section, we will discuss several of them, and describe their inherent differences with our approach.

The basic idea of our approach originates from the concept of *control flow analysis* [25]. Many approaches [26], [27], [28], [29] have been proposed to find errors and bugs in software based on *control flow analysis*, however, only a few of them, such as TinyTracer [30], focus on the WSN applications because of the overhead of tracking the programs' control flow.

TinyTracer is a tracing technique implemented on TinyOS 2.x [31] that records control-flow events of a WSN application program. It can automatically generate highly compressed traces at run-time. Such traces are stored in the flash and can be retrieved later to localize bugs and replay program executions. Since TinyTracer is a control-flow based tracing method, its granularity is limited to the level of basic blocks. Therefore, it is infeasible to reconstruct exact instruction-level history based on this method. For example, if multiple instructions inside a basic block are interleaved with external interrupts, any interleaving order between this basic block and the interrupts will produce exactly the same control-flow traces [30]. Indeed, different from our approach, TinyTracer is not aiming to achieve full-scale instruction level replay.

Different from TinyTracer, AVEKSHA [9] is a combined, hardware-software approach for non-intrusive tracing and profiling of wireless embedded systems. The authors designed a debugging board that interfaces with the on-chip debugging module of an embedded processor through the JTAG interface. The experimental results provided in the AVEKSHA paper showed that AVEKSHA can log a wide range of events for replay purposes. However, the AVEKSHA debugging board is customized using Actel FPGA for the microcontroller unit (MCU) MSP430F1611 from Texas Instruments, which not only adds considerable hardware cost, but also makes it highly platform-dependent.

Finally, a third piece of related work is FSMGen [32]. Strictly speaking, it is not a stand-alone tracing approach. Instead, it is used to infer a user-readable high-level representation of any component of a TinyOS program. This high-level representation accurately extracts system logic from low-level programs, based on an instantiation of the nesC program running on top of the TinyOS operating system, while abstracting away platform-specific details. The authors illustrate that this high-level program representation can be used to aid programmer on understanding, error detection, and correctness validation. However, since this approach is based on a relatively coarse approximation of the TinyOS event-driven execution model, it cannot precisely capture the functionality of low-level interrupt-driven code, such as the timer component and the radio component of TinyOS.

VII. Conclusions and Future Work

In this paper, we have designed and implemented an effective approach that enables logging and replaying of sensor network applications in resource constrained environments. The key insight of this approach is that, by combining runtime traces of sensor network applications together with static analysis of their program structures, we can obtain the entire execution history of applications without the need for additional hardware. We have demonstrated the scalability of this approach with a range of benchmark applications. According to our evaluation results, the presented approach can obtain a high accuracy in replaying original applications at the instruction level, which will help developers obtain better visibility on the actual execution of their sensor network applications. Finally, to demonstrate the usefulness of our approach, we show how to use our approach to reveal non-deterministic program behavior, and detect stack smashing bugs in real sensor network applications.

Although not observed in our experiments, we note that theoretically, it is possible for a developer to carefully craft a piece of code whose execution cannot be completely replayed by our existing approach. For example, such a piece of code may contain a loop, within which there are multiple execution paths. Though we can record how many times each execution path within the loop has been executed, based on the runtime information we collected, our existing approach will not be able to infer the actual execution interleaving of these paths because we only record the newest timestamp for each basic block to reduce tracing overhead. There are two ways to extend our existing framework to address this issue, by either requiring them to change their source code organization to avoid this issue, or adding additional logging points with a higher overhead. We also note that in practice, none of the test cases we tested have met this issue.

In the future, we intend to apply our approach to larger deployments, including those are distributed by nature, and evaluate their corresponding performance. One interesting question we noticed is that although our approach, which is based on using global timing information and execution history for each basic block, proves to be sufficient for individual nodes, is such information sufficient for large-scale, distributed replays? Answering such questions requires us to investigate the unique challenges that arise when the applications span a network of nodes, whose coordination becomes the key to ensure that the results are accurate and dependable.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their insightful comments. The work reported in this paper was supported in part by the National Science Foundation grant CNS-0953238, CNS-1017156, CNS-1117384, and a grant made under the UT-ORNL Joint Directed Research and Development (JDRD) program.

REFERENCES

[1] R. H. B. Netzer and B. P. Miller, "Optimal tracing and replay for debugging message-passing parallel programs," in *Proceedings of the 1992 ACM/IEEE conference Conference on SuperComputing*, 1992.

[2] H. Thane and H. Hansson, "Using deterministic replay for debugging of distributed real-time systems," in *Proceedings of the Euromicro Conference on Real-time Systems*, 2000.

[3] S. Narayanasamy, "Bugnet: Continuously recording program execution for deterministic replay debugging," in *Proceedings of ACM ISCA Conference*, 2005.

[4] D. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay Debugging for Distributed Applications," in *Proceedings of the USENIX Annual Technical Conference*, 2006.

[5] J. Yang and M. Soffa, "Clairvoyant: a comprehensive source-level debugger for wireless sensor networks," pp. 1–15, 2007.

[6] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn, "Respec : Efficient Online Multiprocessor Replay via Speculation and External Determinism," in *Proceedings of ASPLOS*, 2010.

[7] K. H. Lee, N. Sumner, X. Zhang, and P. Eugster, "Unified Debugging of Distributed Systems with Recon," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2011.

[8] R. Chern and K. De Volder, "Debugging with Control-flow Breakpoints," in *Proceedings of the International Conference on Aspect-oriented Software Development (AOSD)*. New York, New York, USA: ACM Press, 2007, p. 96.

[9] M. Tancreti and M. Hossain, "Aveksha: a hardware-software approach for non-intrusive tracing and profiling of wireless embedded systems," in *Proceedings of ACM SenSys*, 2011.

[10] P. Fonseca, C. Li, and R. Rodrigues, "Finding Complex Concurrency Bugs in Large Multi-Threaded Applications," in *Proceedings of EuroSys*, 2011.

[11] B. Kasikci, C. Zamfir, and G. Candea, "Data Races vs . Data Race Bugs: Telling the Difference with Portend," in *Proceedings of ASPLOS*, 2012.

[12] Atmel, "AVR Instruction Set Datasheet," 2010. [Online]. Available: http://www.atmel.com/Images/doc0856.pdf

[13] V. Shnayder, M. Hempstead, B. Chen, G. W. Allen, and M. Welsh, "Simulating the power consumption of large-scale sensor network applications," in *Proceedings of ACM SenSys*. ACM Press, 2004.

[14] G. Necula and S. McPeak, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Proceedings of Compiler Construction*, 2002.

[15] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: scalable sensor network simulation with precise timing," in *Proceedings of the 4th international symposium on Information processing in sensor networks*, 2005.

[16] AVR-LibC, "AVR LibC Library," 2012. [Online]. Available: http://www.nongnu.org/avr-libc/

[17] P. Levis, S. Madden, and J. Polastre, "Tinyos: An operating system for sensor networks," *Ambient Intelligence*, 2005.

[18] T. He, S. Krishnamurthy, L. Luo, T. Yan, L. Gu, R. Stoleru, G. Zhou, Q. Cao, P. Vicaire, J. Stankovic, T. Abdelzaher, J. Hui, and B. Krogh, "VigilNet : An Integrated Sensor Network System for Energy-Efficient Surveillance," *ACM Transactions on Sensor Networks*, vol. 2, no. 1, pp. 1–38, 2006.

[19] Atmel, "ATmega128 Datasheet," 2011. [Online]. Available: http://www.atmel.com/Images/doc2467.pdf

[20] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 15–26.

[21] K. Liu, M. Li, Y. Liu, M. Li, Z. Guo, and F. Hong, "Passive diagnosis for wireless sensor networks," in *Proceedings of the 6th ACM conference on Embedded network sensor systems*, ser. SenSys '08. New York, NY, USA: ACM, 2008, pp. 113–126.

[22] K. Liu, Q. Ma, X. Zhao, and Y. Liu, "Self-diagnosis for large scale wireless sensor networks," in *INFOCOM 2011. 30th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 10-15 April 2011, Shanghai, China*. IEEE, 2011, pp. 1539–1547.

[23] X. Miao, K. Liu, Y. He, Y. Liu, and D. Papadias, "Agnostic diagnosis: Discovering silent failures in wireless sensor networks," in *In Proceedings of IEEE INFOCOM*, 2011.

[24] J. Brown, B. McCarthy, U. Roedig, T. Voigt, and C. Sreenan, "Burst-Probe: Debugging Time-Critical Data Delivery in Wireless Sensor Networks," in *Proceedings of the European Conference on Wireless Sensor Networks (EWSN)*, vol. 6, no. 2, Jan. 2011.

[25] O. Shivers, "Control flow analysis in scheme," 1988, pp. 164–174.

[26] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE TRANSACTIONS ON RELIABILITY*, vol. 51, pp. 111–122, 2002.

[27] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, ser. DFT '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 581–. [Online]. Available: http://dl.acm.org/citation.cfm?id=951947.951989

[28] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 6, pp. 627–641, Jun. 1999. [Online]. Available: http://dx.doi.org/10.1109/71.774911

[29] M. A. Rouf and S. Kim, "Modeling and evaluation of control flow vulnerability in the embedded system," in *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 430–433. [Online]. Available: http://dx.doi.org/10.1109/MASCOTS.2010.71

[30] V. Sundaram, P. Eugster, and X. Zhang, "Efficient Diagnostic Tracing for Wireless Sensor Networks," in *In 8th ACM International Conference on Embedded Networked Sensor Systems*, 2010.

[31] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An Operating System for Sensor Networks Ambient Intelligence," in *Ambient Intelligence*. Springer Berlin Heidelberg, 2005, ch. 7, pp. 115–148.

[32] N. Kothari, T. Millstein, and R. Govindan, "Deriving State Machines from TinyOS Programs using Symbolic Execution," in *Proceedings of ACM IPSN*, 2008, pp. 271–282.