Dynamic Bug Detection in TinyOS
Operating Environments

THESIS
Presented in Partial Fulfillment of the Requirements for
the Degree of Master of Science in the
Graduate School of The Ohio State University

By
Pihui Wei, B.S.

*****

The Ohio State University
2009

Master’s Examination Committee:
Professor Anish Arora, Adviser
Professor Feng Qin

Approved by

Adviser

Computer Science and
Engineering Graduate Program
ABSTRACT

Wireless sensor network applications run on resource scarce sensor nodes. A sensor node usually has extremely limited resources in order to run long time with limited power supply to sense its environment. For example, a XSM[6] sensor node has only 4KB flash memory, a 4MHz processor and a low power radio device.

Sensor network applications are different from computer applications. Most sensor network applications are event driven to run longer with limited power supply. They are in passive mode or sleeping in most of time and can only be activated by some events.

It is very hard to debug sensor network applications because (1) the event driven character of sensor network application makes it hard to predict program behavior; (2) sensor network usually run batched jobs which means people cannot run or debug wireless sensor applications interactively; (3) sensor network application bugs can only be detected after the software is deployed because of the highly distributed and faulty nature of sensor nodes; (4) there is extremely limited visibility to the running program.

This thesis proposes a source-code based dynamic analysis approach to debug wireless sensor network applications. The NesC compiler is modified to automatically insert data collecting code at each program point where program control flow changes. When wireless sensor network applications run on sensor nodes, the reached program point information will be automatically recorded. When program finished, the collected data
can show exactly how program runs. The benefit of this approach is that the data collected are from program product run, it shows the program behavior in real running environment. So the bugs found here are more valuable.
ACKNOWLEDGMENTS

I am thankful to my adviser Prof. Anish Arora for showing me the way to wireless sensor networks area. I am appreciated for all the encouragement and teaching Prof. Anish Arora gave me in the past years. Without his continuous advice, this thesis would not have been possible.

Prof. Feng Qin is a great teacher and great friend of me. I am appreciated for all the helps and advices he gave me during my study in OSU.

I am thankful to Prof. Xiaodong Zhang who gave me great advice and encouragement about research and about life. I am really appreciated for all the helps he gave me.

Thanks Dr. William Leal, Dr. Vinod Kulathumani, Dr. Sandip Bapat, Mr. Taewoo Kwon, Mr. Yan Tang, Mr. Boying Zhang for all the helpful discussions on various projects and topics. I am also thankful to Mr. Mukundan Sridharan who always gave me a hand when I had trouble in using Kansei testbed.
VITA

1999 ................................................................. B. Science, Shandong University

2003 ................................................................. M.S. Computer Science,
Institute of Software,
Chinese Academy of Sciences

FIELD OF STUDY

Major Field: Computer Science and Engineering
Specialization: Computer Networks
TABLE OF CONTENTS

Abstract ................................................................................................................................. ii
Acknowledgments ............................................................................................................. iv
Vita ......................................................................................................................................... v
Table of Contents ............................................................................................................. vi
List of Figures .................................................................................................................. viii
List of Tables ..................................................................................................................... ix
Chapters
1. Introduction ...................................................................................................................... 1
2. Program analysis model and problem definition .......................................................... 4
   2.1 Static analysis .......................................................................................................... 4
   2.2 Dynamic analysis .................................................................................................... 5
   2.3 Problem definition .................................................................................................. 8
3. NesC programming language and TinyOS system ....................................................... 10
   3.1 NesC programming language ................................................................................. 10
   3.2 NesC compiler organization .................................................................................. 12
   3.3 TinyOS system ....................................................................................................... 13
   3.4 TinyOS application compiling ............................................................................... 15
4. Dynamic debugger system architecture .................................................................... 17
   4.1 Architecture ......................................................................................................... 17
   4.2 Data collection code generation rules .................................................................. 19
   4.3 Program point (tracepoint) ID ............................................................................ 24
4.4 Data collecting function implementation ...................................................... 25
4.5 Data analysis ............................................................................................... 27
5. Case studies .................................................................................................... 31
  5.1 SerialReadWriteOnXSM application ............................................................. 33
  5.2 Distributed chowkidar tier one program ....................................................... 38
6. Related works ................................................................................................. 41
  6.1 Declarative tracepoints ................................................................................. 41
  6.2 Sympathy ..................................................................................................... 42
  6.3 Avrora .......................................................................................................... 42
  6.4 t-kernel ....................................................................................................... 43
  6.5 Safe TinyOS ................................................................................................. 44
7. Conclusion and Future work ............................................................................ 45
  7.1 Conclusion ................................................................................................... 45
  7.2 Future work ................................................................................................. 46

REFERENCES ....................................................................................................... 48
LIST OF FIGURES

Figure 1  Serial Read/Write Configuration ................................................................. 12
Figure 2  NesC Application Compiling ........................................................................ 13
Figure 3  TinyOS kernel frame .................................................................................. 15
Figure 4  TinyOS Application Compiling .................................................................... 16
Figure 5  Data Collecting Enabled NesC Compiler Architecture ............................... 19
Figure 6  NesC code conversion .................................................................................. 20
Figure 7  Data Collection Code Generation Rules ...................................................... 23
Figure 8  Sample of data collection code .................................................................... 24
Figure 9  Program Point ID Assignment ..................................................................... 25
Figure 10  Data Collection Function Flowchart ........................................................ 27
Figure 11  Original Point ID Data .............................................................................. 29
Figure 12  Converted Point ID Data .......................................................................... 30
Figure 13  Kansei testbed physical layout ................................................................. 32
Figure 14  A Stargate XSM Pair ................................................................................. 33
Figure 15  Total Number of Collected Program IDs ................................................... 35
Figure 16  Data Collecting Enabled C code for Serial Read/Write test program ......... 37
Figure 17  AMStandard Module code that change value of internal variable 'state' ...... 37
Figure 18  Call chain for UART data transfer .......................................................... 38
Figure 19  Chowkidar tier one program point ID ....................................................... 39
Figure 20  TOS_msg data corruption in HeartBeatM module ..................................... 40
LIST OF TABLES

Table 1 Different devices in Kansei testbed ................................................................. 33
Table 2 Total Number of Collected Program IDs ............................................................ 34
Table 3 Total number of each program ID for Node N105. ........................................... 36
Table 4 Number of point ID numbers collected in Chowkidar tier one program .......... 39
CHAPTER 1

INTRODUCTION

Wireless embedded sensor network[1][2][3] progressed greatly with recent technology improvements. Although originally motivated by military applications such as battlefield surveillance, wireless embedded sensor networks are used widely in industrial and civilian application areas, for example, it is used in monitoring physical or environmental conditions, such as temperature, sound, vibration, pressure, fire, motion, medical condition or pollutants. With more new devices being created, wireless embedded sensor network is becoming a most important area which closely connected to our life.

Wireless sensor network is totally different from traditional Internet. The internet is a global network of connected computers. Every computer has its input/output device and is powerful in computing. Every computer which is connected to the Internet can easily access information and save the information in local memory or file system from other computers or servers easily and can also send information to other computers or servers. Also with several decades of develop, the technology used in the Internet is mature. But for wireless sensor networks are composed of sensor nodes which are usually small, inexpensive, low power devices which can be deployed to any places. Usually there is no display device for sensor nodes. Sensor node has only a limited capability in processing,
has limited power supply. The entire sensor network relies on the coordination of all sensor nodes to finish a task. For example, XSM[4][6] is a common sensor node. It has 4Mbit serial flash memory, can work 1000 hours with 2-AA battery.

It is difficult to debug sensor network applications. Sensor network usually run batched applications. Unlike computer programs which can print messages on screen, there is no or very few visibility for sensor network applications. Because sensor nodes do not have powerful processing capability and input/output devices as computers, also sensor nodes can be deployed to any places to monitor its surrounding environment, it is impossible to run sensor network applications interactively. Usually people send batched sensor network applications to sensor nodes and the applications will run on the nodes continually to coordinate with other nodes. When application starts to run on sensor nodes, people have no control to the running applications.

Another reason that sensor network application is difficult to debug is that sensor network applications are usually event driven applications. Different event will trigger different program module. It is hard to predict the program control flow statically because of the event driven nature.

This thesis describes a dynamic approach to detect bugs in sensor network applications. By running data collecting enabled applications on real sensor nodes, lots of programming running information data will be collected and saved in a log file. An offline analysis will be applied to the log data to find potential bugs.

The rest of the thesis is organized as follows: Chapter 2 describes dynamic bug detection model and problem definition; Chapter 3 describes NesC[7] programming
language and TinyOS[8] computing environment; Chapter 4 describes dynamic bug detection system architecture; Chapter 5 describes case studies and system evaluation; Chapter 6 describes related works and Chapter 7 concludes this thesis and introduces future work in this direction.
CHAPTER 2

PROGRAM ANALYSIS MODEL AND PROBLEM

DEFINITION

Static analysis [9] and dynamic analysis are two analysis methods to analyze computer programs. This chapter describes basic static analysis approaches and dynamic analysis model used in this thesis. Then define the problem of dynamically detecting program bugs in TinyOS computing environment.

2.1 STATIC ANALYSIS

Static analysis is used to analyze programs by using compile-time information. By analyzing some form of source code, static analysis tool can generate some static models of all possible run-time behavior of the analyzed program without actually running it. The static model can either be generated manually or automatically. Call graph is a well studied model of programs. NesC[7] compiler automatically generates a call graph from NesC source code and finally generate an intermediate C code of NesC code. When a static model is generated, there are many approaches to analyze the model. Basically there are two kinds of static model checkers[10]: model checker[12] and theorem prover.
Model checker is a technique to operate directly on static program model to study the program control flow, data flow or inter-procedure call flow, etc. For example, JPF[14] is a verification and testing environment for Java. It extended Java Virtual Machine to interpret Java bytecode generated by Java compiler.

Theorem prover is to use pure mathematical formula or logic formula to analyze program model. It usually requires translating program into logic formula, then applying theorem prover on the logic formula to find program errors. ESC/Java[13] uses an automatic theorem prover to reason about the semantics of programs and give static warnings about many errors that are caught at runtime by modern programming languages.

NesC compiler has an integrated static analyzer to analyze the call graph generated from preprocessor stage. Chapter 3 describes the NesC compiler in detail. NesC compiler will detect race conditions based on how shared variables used.

2.2 DYNAMIC ANALYSIS

Dynamic analysis is performed by executing computer software on real processors or virtual machines. It operates on information collected from the program at run time. By gathering information from software product run, dynamic analysis can get accurate information how program actually runs and how the program state changes. Dynamic analysis can be used for program error detection, error correction, optimization and program understanding [15].

Dynamic analysis can be distinguished as source code based dynamic analysis and binary code based dynamic analysis. In source code based dynamic analysis, source code
is thoroughly analyzed and modified to output language information. For example, Xu[16] gave an C program transformation approach to ensure memory safety of C program. In this approach, they modify C source to enhance every memory pointer with a metadata structure. With this data structure they can precisely control and observe the access to that pointer. The advantages of source code based approach are (1) by working on source code, it is convenient to add different code to collect various kind of language level information; (2) although adding additional codes in to program source code means increasing program size, compiler optimization can dramatically decrease the overhead of the additional code; (3) it is easy to create common tools to handle all program source codes of a specific language. But unfortunately, source code based approach has its own drawbacks (1) it is a challenge to build program from source code for big applications, especially when there are plenty of platform dependent control in the source code; (2) For some language, it is not easy to create good common tools to handle all source codes.

Binary code based approach works on program binary code to collect program running information and/or observe program behavior. A well designed binary code based program analysis tool can provide a virtual machine or virtual running environment to run binary code without requiring any change to the program itself. Purify[18] is a software testing and quality assurance tool that detect memory leaks and memory access violation. It works on program binary code by inserting additional checking instructions to check every memory read and write. It inserts checking instructions to all program binary code including third-party shared library code. So it can check memory access on the whole program. Binary code based program analysis has the following advantages: (1) It can test complete program instead of only a part of program. Because it works on
binary level and usually it provides virtual machine to simulate the running environment, it can easily get access to every bit of a program, including third-party library code. (2) It does not require any change to program source code or binary code. But binary code based dynamic analysis also has some difficulties: (1) It is usually hard to provide a virtual machine or virtual running environment for being tested program. The virtual machine or virtual running environment need to simulate hardware and software environment for binary program. (2) It is impossible to provide a common dynamic analysis tool for all platforms because the difference between different platform architecture.

TinyOS[20] is the dominating systems for programming wireless sensor networks for long time. TinyOS itself and TinyOS applications are programmed in NesC language. Professor Atanas Rountev has proposed many possible approaches to debug NesC programs. Static analysis and dynamic analysis can both be used to NesC program. In static analysis, call graph, interprocedural control-flow graph can be used. In dynamic analysis, many run-time measurements can be used, for example, the number of times a function is entered/exited, the number of times that program control flow goes from a function to another function, the number of times a call chain (a chain of adjacent edges) is followed forward or backward, etc.

This thesis describes a source code based dynamic analysis approach for TinyOS application for the following reasons:
(1) NesC language is a simplified C dialect language. It is used for sensor nodes with limited resources. It is very simple and easy to be rebuilt when there is any change to its source code.

(2) Embedded sensor network applications are usually small applications. The logic of the program is not as complicated as big C/C++ computer applications.

(3) Source code based dynamic analysis tool can easily generate application binary code for different platforms. The gcc compiler will handle platform differences.

(4) NesC compiler’s optimization can decrease the overhead added by inserted data collection code.

2.3 PROBLEM DEFINITION

The source-code based dynamic analysis tool in this thesis is used to detect program deficiencies in sensor network applications, especially the bugs in TinyOS kernel. Chapter 4 will describe source code based dynamic analysis in detail. The problem is defines as follows:

“For a set of $n$ homogeneous sensor nodes $\{N_1, N_2, N_3, \ldots, N_n\}$, run the same data collecting enabled TinyOS application on every node. The applications collect enough program behavior data $\{D_1, D_2, \ldots, D_n\}$ on each node. Then compare and analysis the collected data from different nodes to find bugs in TinyOS application itself or in TinyOS kernel”.

In this thesis, the program behavior data collected are IDs of program points that the program control flow reaches. In NesC program source code, on every statement that can
cause program control flow change, a new program point ID will be created and a statement to collect the point ID will be inserted. When program control reaches that point, the point ID will be collected and sent to data collector on Stargate. Based on the data collected, three kinds of application running information differences are examined, total number of collected point IDs, number of a specific point ID and program control flow difference.
CHAPTER 3

NESC PROGRAMMING LANGUAGE AND TINYOS SYSTEM

TinyOS[20] is the dominating systems for programming wireless sensor networks for long time. A TinyOS application is a collection of components written in NesC[21] with some additional runtime support. When writing a TinyOS application, a programmer usually writes a collection of custom modules and link these modules with TinyOS modules. This chapter describes TinyOS and NesC architecture which is closely related to this thesis.

3.1 NESC PROGRAMMING LANGUAGE

As a extension of C language, NesC is designed to embody the structuring concepts and execution model of TinyOS[8][20]. In NesC, interface and component are the two basic items. In NesC, interface defines the behavior of components. Interface provides definition of commands and events. A command is a kind of function need to be present in the application to do some useful thing. An event defines the actions to be triggered when a command is finished or some hardware event happened. For example, TinyOS kernel provided a ReadData interface (defined in ReadData.nc file). The command
“read” is defined as “command result_t read(uint32_t offset, uint8_t* buffer, uint32_t numBytesRead)” to read data. ReadData interface also defines an event readDone as “event result_t readDone(uint8_t* buffer, uint32_t numBytesRead, result_t success)” which will be triggered when the “read” command is finished. An interface can be used or provided by components. When a component provides an interface, it needs to provide the implementation of that interface’s commands. When a component uses an interface, it needs to provide the implementation of that interface’s event. By this directional characteristic, an interface can represent a complex interaction between components.

A typical NesC application is composed of components. There are two parts in components: its specification and its implementation. There are two kinds of components: modules and configurations. All components are wired together by configurations to form the whole application. For example, Figure 1 [22] is the configuration code of SerialTest[22], which is a simple TinyOS application being used to collect XSM sensor data every second and send to data to Stargate in Kansei testbed[23]. Configuration implementation part defines how components are wired with each other (e.g. Main.StdControl is wired as SerialTestM.StdControl).

Module is the place where commands or events are implemented. When a component provides an interface, it needs to implement the interface’s commands. When a component uses an interface, it needs to implement the interface’s event. Some events are triggered when the corresponding command are finished. Some primitive events are triggered when some hardware event happened.
includes crc;

classification SerialTest {
}
implementation {
    components Main, SerialTestM, TimerC, LedsC, GenericComm;

    Main.StdControl -> SerialTestM.StdControl;
    Main.StdControl -> GenericComm;
    Main.StdControl -> TimerC;

    //SerialTestM.CommControl -> GenericComm;
    SerialTestM.UARTSend -> GenericComm.SendMsg[100];
    SerialTestM.UARTReceive -> GenericComm.ReceiveMsg[99];
    SerialTestM.Leds -> LedsC;
    SerialTestM.Timer -> TimerC.Timer[unique("Timer")];
}

Figure 1 Serial Read/Write Configuration

3.2 NESC COMPILER ORGANIZATION

NesC compiler is a set of programs to convert a NesC program into a binary program that can be run on sensor nodes. Figure 2 shows the four parts of NesC compiler: Preprocessor, parser&builder, C code generator and C compiler.

Preprocessor does some initial preprocess to the NesC application source code. NesC program can contains three kinds of source code files: C header file(.h file), C source file(.c file) and NesC file(.nc file). The preprocessor will reorganize the source files to a uniform format to be processed by parser.

Parser and Builder is the part to parse the source code and generate call graph. The main function of this part is defined in c-parser.y file which defines NesC grammar. The
generated call graph is used to generate c file and can also be used to do data race condition static checking.

C code generator works on call graph to generate c file (app.c file).

At last, C compiler will compile the c file into binary code which can be run on sensor nodes.

![Diagram of NesC Application Compiling]

**Figure 2** NesC Application Compiling

### 3.3 TINYOS SYSTEM

TinyOS[8][20] is a dominant system for programming wireless sensor network devices. Usually wireless sensor network devices have extremely limited resources, e.g. small size, small amount of flash as program memory, limited power supply etc. But wireless sensor network applications usually need to be run for long time. For example, an environment
monitoring sensor network application needs to be run for months to monitor the environment change. TinyOS is designed to support applications to run on sensor devices with limited resources and to run for long time without error.

A TinyOS application is essentially a graph of components which are wired together by wire specifications defined in configurations. A programmer usually writes some custom components which are specific to the application and link these components with system components from TinyOS. TinyOS provides a large library of well designed system components to support a big amount of popular sensor devices, so people can save much of work in developing their own TinyOS applications.

TinyOS has a task and event-based concurrency [7] mechanism. An important goal of TinyOS is to handle large number of current data flows, outstanding events and to manage application data processing. In TinyOS, tasks and events are the two sources of concurrency. Tasks are deferred computing mechanism. There is a task queue in TinyOS. When a component posts a task, the task will be put in the task queue. Later the TinyOS kernel will run all tasks in queue when it gets a chance. Event will be triggered when some hardware or software event happens. Both task and event are run to completion. But events can preemptive tasks or other events and tasks cannot preemptive other tasks or events.

TinyOS has a low duty-cycle to conserve power. Figure 3[24] shows the TinyOS main frame which is generated from NesC compiler. After hardware and application initialization, TinyOS application goes in an infinite loop which will periodically pick all tasks in task queue to run and then goes in asleep to save power.
int main(void)
{
    RealMain$hardwareInit();
    RealMain$Pot$init(10);
    TOSH_sched_init();
    RealMain$StdControl$init();
    RealMain$StdControl$start();
    __nesc_enable_interrupt();
    while (1)
    {
        TOSH_run_task();
    }
}

static inline void
TOSH_run_task()
{
    while (TOSH_run_next_task())
    {
        ;
    }
    TOSH_sleep();
    TOSH_wait();
}

bool TOSH_run_next_task(void)
{
    __nesc_atomic_t fInterruptFlags;
    uint8_t old_full;
    void (*func)(void);
    fInterruptFlags = __nesc_atomic_start();
    old_full = TOSH_sched_full;
    func = TOSH_queue[old_full].tp;
    if (func == (void *)0)
    {
        __nesc_atomic_end(fInterruptFlags);
        return 0;
    }
    TOSH_queue[old_full].tp = (void *)0;
    TOSH_sched_full = (old_full + 1) & TOSH_TASK_BITMASK;
    __nesc_atomic_end(fInterruptFlags);
    func();
    return 1;
}

Figure 3 TinyOS kernel frame

3.4 TINYOS APPLICATION COMPILING

The compiling of TinyOS application is the same as the compiling of NesC application introduced in section 3.2. Figure 4 shows the compiling process of a TinyOS application. The dynamic analysis tool in this thesis works on the call graph generated by parser & builder (see figure 2) and inject data collection stubs to record program behavior. It takes care of both programmer created components and TinyOS library components. The reason is that: The approach focus on the logic of the program and try to record how the program runs. It does not care about the definition of configurations, interfaces because when the call graph was generated, all these information disappear.
Figure 4 TinyOS Application Compiling
CHAPTER 4

DYNAMIC DEBUGGER SYSTEM ARCHITECTURE

This chapter describes the architecture and the implementation of the analysis tool. Then the approach to analysis collected data will be described.

4.1 ARCHITECTURE

The basic idea of the dynamic program analysis in this thesis is to insert data collecting code to TinyOS application program. The code will collect data automatically when the program runs on wireless sensor devices. The data collected are point ID numbers which represent positions in program. The collected point IDs show the control flow of the running program.

Figure 5 shows the architecture of the new NesC compiler that can compile TinyOS application and generate program with data (program point ID) collecting capability. There is a new “Data Collection Rules” module added to the compiler to help “C Code Generator” generate C code to collect data when program runs on sensor devices.
There are two possible places where the “Data Collection Rules” module can be applied to: the NesC grammar definition (in file c-parser.y file of NesC compiler source code) or the “C Code Generator”. In this thesis, the data collection rules are applied to C code generator because:

(1) NesC syntax definition can not affect functions defined in C code. TinyOS application contains NesC code and/or C code. Programmer can write some subroutines in C code and call the subroutines in NesC code and/or C code. The subroutines definition cannot be seen during NesC code preprocessor and Parser phase. So there is no way to insert data collection code to functions defined in C code.

(2) There is no enough function information collected during Parser&Builder part stage. The parser& Builder phase is a recursive phase to process NesC code. Before the call graph is fully built, the function definition is not complete. But some functions are special and there should be no data collection code inserted, for example, the data collection code need to call UARTSend.send() function to send data to Stargate, there should be no data collection code in UARTSend.send() function body in order to avoid infinite function call loop.

When the call graph is fully built, the compiler will call “C Code Generator” to generate C code from the call graph. The modified NesC compiler (Figure 5) will consult the “Data Collection Rules” to insert data collection code for the current function.
4.2 DATA COLLECTION CODE GENERATION RULES

Data collection code is used to record program point and send the point ID data to Stargate. On the Stargate side serial forwarder program will write data to a log file. The point ID is used to record program control flow. A new point ID will be assigned when a new data collection function call is inserted to the generated C file.

The C Code Generator will scan the whole call graph, specifically all modules in the call graph and link all the modules together to create a single C file that contains all code. In the call graph, each module has a private data area, a private functions definition area and public interface definition area. All the function calls, data type name and data references are converted to full qualified name, containing module name and function name, data type name or variable name. For example, Figure 6 shows code conversion...
from NesC code to C code. C Code Generator converts the data type name test_msg to SerialTestM$test_msg, converts the variable name doTrack to SerialTestM$doTrack and converts the command name StdControl.init() to SerialTestM$StdControl$init(). The module codes (command function code or event code) are essentially unchanged.

Figure 6 NesC code conversion

In the new modified NesC compiler, C Code Generator will insert data collection code to collect data when program runs. Figure 7 shows the Data Collection Rules the C Code Generator will use. The following places will be considered to insert data collection code:

(1) Entrance to function body.
(2) Conditional statement:

a. For “if (expression) Stmt”, the entrance to the compound statement Stmt will be recorded.

b. For “if(expression) Stmt1 else Stmt2”, the entrance to the compound statements Stmt1 and Stmt2 will be recorded.

(3) Label statement: A data collection code will be placed right after the label, so when program control comes to label point by goto statement, the point will be recorded.

(4) Loop statement: the entrance to loop statement body will be recorded.

(5) Switch statement: Switch statement has many branches based on the tag value, so a data collection statement will be inserted to each branch.

(6) Function call: Function call can cause program control flow change, so a data collection statement will be placed right before the function call.

(7) Break statement: break statement will make program control flow leave a loop body, so a data collection statement will be placed right before the break statement.

(8) goto statement: goto statement will change program control flow to a label statement, so a data collection statement will be placed right before the goto statement.

Data collecting function will be disabled in atomic statement block. Atomic statement in NesC language has the following form:

atomic statement; or atomic {compound statement};
By disabling interrupts, atomic statement stops events from being signaled directly or indirectly by interrupts. NesC requires that atomic statement should be finished very quickly. If an atomic takes too much time to finish, there will be a lot of interrupts, essentially events, being lost. Data collecting statement includes function call to send collected point ID data to Stargate. The data sending function will take long time, so there will be no data collection statement inserted to atomic statement.

By applying data collecting rules to C Code Generator, C code as in Figure 8 can be generated and be written in file app.c.
When C Code Generator is outputing statement S,
if S is an “If (exp) then Stmt” then
  Write an data collection statement right before the first statement of compound
  statement Stmt;
else if S is an “If (exp) then Stmt1; else Stmt2” then
  Write an data collection statement right before the first statement of compound
  statementStmt1 and Stmt2;
else if S is a “Label” then
  Write an data collection statement right after the label;
else if S is a “Switch statement” then
  Write an data collection statement right after every “Case expr:” condition;
else if S is a “do Stmt while(Expr)” then
  Write an data collection statement right before the first statement of compound
  statement Stmt;
else if S is a “While(expr) Stmt” then
  Write an data collection statement right before the first statement of compound
  statement Stmt;
else if S is a function call then
  Write an data collection statement right before and after the function call;
else if S is a “break” statement then
  Write an data collection statement right before the break statement;
else if S is a “goto” statement then
  Write an data collection statement right before the goto statement;
else if S is a “for (…) stmt” statement,then
  Write an data collection statement right before the first statement of compound
  statement Stmt;
else if S is an “atomic { Stmt }” statement then
  Write a data collection statement right before S and disable any data collection in
  compound statement Stmt
else
  Do nothing;

Figure 7 Data Collection Code Generation Rules
4.3 PROGRAM POINT(TRACEPOINT) ID

Every time when C Code Generator inserts a data collecting code, a new program point(TracePoint) ID is generated and assigned to that program point. All the point ID information will be written to a separate file. When program runs on XSM, only point ID numbers are recorded and sent to serial forwarder program on Stargate. Figure 9 shows a segment of the program point ID list file created for SerialReadWriteTest program.

When a new point ID created, three information will be saved: point ID, file name and code line number. File name means in which source code file the program point is created.
Code line means at which source code (TinyOS library source code or application source code) line is the point ID created.

Call graph contains module definitions and/or function definitions for all modules used in TinyOS application, both from programmer created source code and TinyOS library source code. Usually there are several hundred to several thousand point IDs created for a TinyOS application.

<table>
<thead>
<tr>
<th>Position ID</th>
<th>File name</th>
<th>Code Line #</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>C:/PROGRA~1/UCB/cygwin/opt/tinyos-1.x/tos/system/PotM.nc</td>
<td>108</td>
</tr>
<tr>
<td>62</td>
<td>C:/PROGRA~1/UCB/cygwin/opt/tinyos-1.x/tos/interfaces/Pot.nc</td>
<td>78</td>
</tr>
<tr>
<td>63</td>
<td>C:/PROGRA~1/UCB/cygwin/opt/tinyos-1.x/tos/system/sched.c</td>
<td>84</td>
</tr>
<tr>
<td>64</td>
<td>C:/PROGRA~1/UCB/cygwin/opt/tinyos-1.x/tos/system/sched.c</td>
<td>89</td>
</tr>
<tr>
<td>65</td>
<td>C:/PROGRA~1/UCB/cygwin/opt/tinyos-1.x/tos/system/tos.h</td>
<td>124</td>
</tr>
<tr>
<td>66</td>
<td>C:/PROGRA~1/UCB/cygwin/opt/tinyos-1.x/tos/system/tos.h</td>
<td>125</td>
</tr>
<tr>
<td>67</td>
<td>C:/PROGRA~1/UCB/cygwin/opt/tinyos-1.x/tos/interfaces/StdControl.nc</td>
<td>63</td>
</tr>
<tr>
<td>68</td>
<td>C:/PROGRA~1/UCB/cygwin/opt/tinyos-1.x/tos/system/RandomLFSR.nc</td>
<td>59</td>
</tr>
<tr>
<td>69</td>
<td>C:/PROGRA~1/UCB/cygwin/opt/tinyos-1.x/tos/system/RandomLFSR.nc</td>
<td>66</td>
</tr>
<tr>
<td>70</td>
<td>C:/PROGRA~1/UCB/cygwin/opt/tinyos-1.x/tos/system/RandomLFSR.nc</td>
<td>57</td>
</tr>
<tr>
<td>71</td>
<td>C:/PROGRA~1/UCB/cygwin/opt/tinyos-1.x/tos/platform/xsm/HPLADCM.nc</td>
<td>90</td>
</tr>
<tr>
<td>72</td>
<td>C:/PROGRA~1/UCB/cygwin/opt/tinyos-1.x/tos/platform/xsm/HPLADCM.nc</td>
<td>100</td>
</tr>
<tr>
<td>73</td>
<td>C:/PROGRA~1/UCB/cygwin/opt/tinyos-1.x/tos/interfaces/HPLADC.nc</td>
<td>54</td>
</tr>
<tr>
<td>74</td>
<td>C:/PROGRA~1/UCB/cygwin/opt/tinyos-1.x/tos/platform/mica2/ADCREFM.nc</td>
<td>99</td>
</tr>
<tr>
<td>75</td>
<td>C:/PROGRA~1/UCB/cygwin/opt/tinyos-1.x/tos/platform/mica2/ADCREFM.nc</td>
<td>107</td>
</tr>
<tr>
<td>76</td>
<td>C:/PROGRA~1/UCB/cygwin/opt/tinyos-1.x/tos/platform/mica2/ADCCcontrol.nc</td>
<td>77</td>
</tr>
</tbody>
</table>

Figure 9 Program Point ID Assignment

4.4 DATA COLLECTING FUNCTION IMPLEMENTATION

The data collecting code is a new function defined to record and send point ID data to Stargate’s serial forwarder program. It is defined as:

```c
static void sendPacketToStargate(int start_point_tracker1, uint16_t pointNum);
```
The parameter “start_point_tracker1” is a Boolean variable to enable/disable data collecting function. When this value is true, function sendPacketToStargate() will collect data, or function sendPacketToStargate() will not collect data. The parameter “pointNum” is the point ID number which is used to uniquely represent the current program point. The point ID is a 16-bit unsigned integer value. The system can create 65536 new point IDs at most which is enough for most TinyOS applications.

A data buffer of 200 16-bit integer values is defined in the final C code file to cache collected program point ID numbers. The buffer is used here because the data channel from sensor node to Stargate maybe busy when data collecting routine wants to transfer data. The new collected data can be put in cache to be transferred when the channel is free. The buffer is defined as:

```
#define __TRK_BUFFER_SIZE 200
// need a buffer to cache data
uint16_t trk_buffer[__TRK_BUFFER_SIZE];
```

Figure 10 shows flowchart of data collecting function. Every time it sends out 14 ubit16_t value. The first one is a tag to mark this group of data is point IDs collected when program runs. The others 13 values are the 13 point IDs collected.
Figure 10 Data Collection Function Flowchart

4.5 DATA ANALYSIS

Figure 11 shows the originally collected data which is sent from sensor node to Stargate node. Data in Figure 11 are converted to data in Figure 12 which contains the decimal value of program point IDs. In Figure 12, the number 1121 means there are totally 1121 point IDs created when the modified NesC compiler compiles application source code. The number 1901393 is the total number of point IDs collected when the TinyOS programs runs on sensor device. The “Program ID Sequence” section contains
the entire program IDs collected. From the data in “Program ID Sequence” section the following information can be generated:

(1) Program control flow:

Because first collected program ID data will be sent to Stargate early, the sequence of the program IDs in Figure 12 shows how the program control flow goes. From this information, it is easy to generate call chains information.

(2) Reach times of every program point:

Apparently the reach times of each program ID can be easily counted from the data in Figure 12.

In this thesis, collected data will be analyzed from three aspects:

(1) By checking the difference of total point IDs collected by every sensor device, it is easy to find which node is the best to be carefully studied in order to find bugs.

(2) By checking the difference of total number of a specific point ID collected by every sensor device, it is easy to decide which program point is the best point to start to search for bugs.

(3) By checking the difference of control flows defined by collected point ID data, it is easy to find which segment of part is the reason of the abnormality.
Figure 11 Original Point ID Data
Program ID Sequence:

<table>
<thead>
<tr>
<th>678</th>
<th>65</th>
<th>66</th>
<th>65</th>
<th>66</th>
<th>70</th>
<th>71</th>
<th>65</th>
<th>66</th>
<th>73</th>
</tr>
</thead>
<tbody>
<tr>
<td>74</td>
<td>72</td>
<td>677</td>
<td>90</td>
<td>675</td>
<td>676</td>
<td>89</td>
<td>87</td>
<td>88</td>
<td>96</td>
</tr>
<tr>
<td>94</td>
<td>679</td>
<td>95</td>
<td>93</td>
<td>91</td>
<td>92</td>
<td>157</td>
<td>154</td>
<td>155</td>
<td>155</td>
</tr>
<tr>
<td>155</td>
<td>155</td>
<td>155</td>
<td>155</td>
<td>155</td>
<td>155</td>
<td>155</td>
<td>155</td>
<td>155</td>
<td>155</td>
</tr>
<tr>
<td>148</td>
<td>146</td>
<td>145</td>
<td>143</td>
<td>32</td>
<td>142</td>
<td>30</td>
<td>31</td>
<td>141</td>
<td>140</td>
</tr>
<tr>
<td>139</td>
<td>144</td>
<td>120</td>
<td>680</td>
<td>158</td>
<td>681</td>
<td>682</td>
<td>684</td>
<td>159</td>
<td>160</td>
</tr>
<tr>
<td>139</td>
<td>681</td>
<td>682</td>
<td>684</td>
<td>159</td>
<td>160</td>
<td>139</td>
<td>681</td>
<td>682</td>
<td>684</td>
</tr>
<tr>
<td>159</td>
<td>160</td>
<td>139</td>
<td>681</td>
<td>682</td>
<td>684</td>
<td>159</td>
<td>160</td>
<td>160</td>
<td>398</td>
</tr>
<tr>
<td>682</td>
<td>684</td>
<td>159</td>
<td>160</td>
<td>139</td>
<td>681</td>
<td>682</td>
<td>684</td>
<td>159</td>
<td>160</td>
</tr>
<tr>
<td>139</td>
<td>681</td>
<td>682</td>
<td>684</td>
<td>159</td>
<td>160</td>
<td>139</td>
<td>140</td>
<td>160</td>
<td>139</td>
</tr>
<tr>
<td>141</td>
<td>685</td>
<td>686</td>
<td>688</td>
<td>159</td>
<td>160</td>
<td>139</td>
<td>685</td>
<td>686</td>
<td>688</td>
</tr>
<tr>
<td>159</td>
<td>160</td>
<td>139</td>
<td>685</td>
<td>686</td>
<td>687</td>
<td>140</td>
<td>160</td>
<td>139</td>
<td>685</td>
</tr>
<tr>
<td>686</td>
<td>687</td>
<td>140</td>
<td>160</td>
<td>139</td>
<td>685</td>
<td>686</td>
<td>687</td>
<td>140</td>
<td>160</td>
</tr>
<tr>
<td>139</td>
<td>685</td>
<td>686</td>
<td>688</td>
<td>159</td>
<td>160</td>
<td>139</td>
<td>685</td>
<td>686</td>
<td>687</td>
</tr>
<tr>
<td>140</td>
<td>160</td>
<td>139</td>
<td>685</td>
<td>686</td>
<td>687</td>
<td>159</td>
<td>160</td>
<td>139</td>
<td>141</td>
</tr>
<tr>
<td>140</td>
<td>139</td>
<td>689</td>
<td>120</td>
<td>680</td>
<td>158</td>
<td>681</td>
<td>682</td>
<td>684</td>
<td>159</td>
</tr>
<tr>
<td>160</td>
<td>139</td>
<td>681</td>
<td>682</td>
<td>684</td>
<td>159</td>
<td>160</td>
<td>139</td>
<td>681</td>
<td>682</td>
</tr>
<tr>
<td>684</td>
<td>159</td>
<td>160</td>
<td>139</td>
<td>681</td>
<td>682</td>
<td>684</td>
<td>159</td>
<td>160</td>
<td>139</td>
</tr>
<tr>
<td>681</td>
<td>682</td>
<td>684</td>
<td>159</td>
<td>160</td>
<td>139</td>
<td>681</td>
<td>682</td>
<td>684</td>
<td>159</td>
</tr>
<tr>
<td>160</td>
<td>139</td>
<td>681</td>
<td>682</td>
<td>684</td>
<td>159</td>
<td>160</td>
<td>139</td>
<td>140</td>
<td>160</td>
</tr>
<tr>
<td>139</td>
<td>141</td>
<td>685</td>
<td>686</td>
<td>688</td>
<td>159</td>
<td>160</td>
<td>139</td>
<td>685</td>
<td>686</td>
</tr>
<tr>
<td>688</td>
<td>159</td>
<td>160</td>
<td>258</td>
<td>252</td>
<td>253</td>
<td>256</td>
<td>258</td>
<td>252</td>
<td>634</td>
</tr>
<tr>
<td>253</td>
<td>254</td>
<td>634</td>
<td>255</td>
<td>634</td>
<td>674</td>
<td>258</td>
<td>252</td>
<td>253</td>
<td>357</td>
</tr>
<tr>
<td>358</td>
<td>358</td>
<td>358</td>
<td>358</td>
<td>358</td>
<td>358</td>
<td>358</td>
<td>358</td>
<td>358</td>
<td>258</td>
</tr>
<tr>
<td>252</td>
<td>253</td>
<td>256</td>
<td>258</td>
<td>252</td>
<td>253</td>
<td>254</td>
<td>254</td>
<td>634</td>
<td>255</td>
</tr>
<tr>
<td>634</td>
<td>674</td>
<td>258</td>
<td>252</td>
<td>253</td>
<td>256</td>
<td>258</td>
<td>252</td>
<td>634</td>
<td>253</td>
</tr>
<tr>
<td>254</td>
<td>634</td>
<td>255</td>
<td>634</td>
<td>674</td>
<td>258</td>
<td>252</td>
<td>253</td>
<td>256</td>
<td>258</td>
</tr>
<tr>
<td>252</td>
<td>634</td>
<td>253</td>
<td>254</td>
<td>255</td>
<td>634</td>
<td>634</td>
<td>674</td>
<td>258</td>
<td>252</td>
</tr>
</tbody>
</table>

Figure 12 Converted Point ID Data
CHAPTER 5

CASE STUDIES

This chapter describes two experiments made on two TinyOS applications: the SerialReadWriteOnXsm[22] and the Chowkidar tier one[5] application running on Kansei[19][23] testbed. The kansei testbed provides a testbed infrastructure to conduct experiments with 802.11b networking and XSMs. In Kansei testbed, there are 210 Extreme Scale Motes (XSM) which are hooked individually onto 210 Extreme Scale Stargates (XSS). Figure 13 shows the physical layout of Kansei testbed. The stargates are connected using both wired and wireless Ethernet and on each Stargate there is a XSM plugged in its serial port. Figure 14 shows the Stargate-XSM pair. All XSMs can communicate with others XSMs via radio connection. Table 1[5] lists all Kansei devices and characters.

The two applications are compiled with modified NesC compiler to generate data collecting enabled binary code to run on XSM. For each application, the compiler generates a configure file “EncodeFileConfigureOutput.txt” which contains all the
program point ID configuration as shown in Figure 9. For each program point ID, it is easy to target the unique program point in TinyOS application or kernel source code. After running application on XSM, a log file is created for each XSM node to record collected program point IDs. The analysis are based on three parts of information: (1) Collected program point IDs for each XSM; (2) The program point ID configure file “EncodeFileConfigureOutput.txt” for each XSM; and (3) NesC application and TinyOS kernel source code.

In this chapter, a bug is identified when a node has significant less of data collected compared to other nodes’ collected data. The reason is that TinyOS program has an infinite loop and data should be collected all the time. If we observed that a node stopped collecting data, there must be something wrong with the program running on the sensor node.

![Figure 13 Kansei testbed physical layout](image)
Figure 14 A Stargate XSM Pair

<table>
<thead>
<tr>
<th>Device Type</th>
<th>XSM</th>
<th>TelosB</th>
<th>Stargate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>4MHz</td>
<td>8MHz</td>
<td>400MHz</td>
</tr>
<tr>
<td>RAM</td>
<td>4KB</td>
<td>10KB</td>
<td>32MB</td>
</tr>
<tr>
<td>OS</td>
<td>TinyOS</td>
<td>TinyOS</td>
<td>Linux</td>
</tr>
<tr>
<td>Interfaces</td>
<td>CC1000, Serial</td>
<td>CC2420, USB</td>
<td>Ethernet, 802.11b, Serial, USB</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>38.4Kbps</td>
<td>250Kbps</td>
<td>11Mbps</td>
</tr>
</tbody>
</table>

Table 1 Different devices in Kansei testbed

5.1 SERIALREADWRITEONXSM APPLICATION

This is a simple TinyOS application used to test the XSM-stargate interface. It sends a packet from XSM to Stargate every second. Based on the Data Collecting Code Generation Rules described in Chapter 4, the new NesC compiler creates 849 program
point IDs to record all possible program control flow changes in the final generated C code file.

Table 2 shows the total number of program IDs collected by the application running on XSM (2 rounds) for 3 hours. Figure 15 shows the bar chart of the total program IDs collected. From both Table 2 and figure 15 we can see:

(1) N53, N59 and N180 collected more than 240K in both rounds.

(2) N173 and N263 collected less than 70K data in both rounds.

(3) N105 behaved very special. In round 1, it collected 584K data, but in round 2, it collected 32K data.

<table>
<thead>
<tr>
<th>XSM Nodes</th>
<th>Round 1</th>
<th>Round 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>N53</td>
<td>613223</td>
<td>242632</td>
</tr>
<tr>
<td>N59</td>
<td>290940</td>
<td>257478</td>
</tr>
<tr>
<td>N65</td>
<td>180193</td>
<td>18395</td>
</tr>
<tr>
<td>N67</td>
<td>83746</td>
<td>153166</td>
</tr>
<tr>
<td>N96</td>
<td>215150</td>
<td>300963</td>
</tr>
<tr>
<td>N105</td>
<td>584675</td>
<td>32591</td>
</tr>
<tr>
<td>N173</td>
<td>6656</td>
<td>24258</td>
</tr>
<tr>
<td>N180</td>
<td>359658</td>
<td>762736</td>
</tr>
<tr>
<td>N194</td>
<td>103766</td>
<td>102479</td>
</tr>
<tr>
<td>N218</td>
<td>194064</td>
<td>383955</td>
</tr>
<tr>
<td>N233</td>
<td>88764</td>
<td>42562</td>
</tr>
<tr>
<td>N236</td>
<td>61009</td>
<td>32773</td>
</tr>
</tbody>
</table>

Table 2 Total Number of Collected Program IDs
Bug #1: Task queue overflow

This bug is generated by analyzing XSM N105. For node N105, the second run has significant fewer collected data than the first run. Table 3 lists total collected data by XSM N105 for function TOSH_run_task() and TOSH_run_next_task(). The source code of these two functions is described in Figure 16.

From the collected program point ID data, we can get the program control flow and the program function call chain. By studying the program function call chain and control flow in round 2, we found the TinyOS kernel function AMStandard$reportSendDone was not called when the collected data transfer from XSM mote to Stargate was finished. But in round 1, the function appeared regularly once after a while. Further study to the source
shows that the AMStandard$reportSendDone() function is critical because it will change the value of AMStandard$state to FALSE so later the program will be able to send data from XSM to Stargate. Figure 17 show the source codes to manage the module internal variable ‘state’ which is compiled as AMStandard$state by NesC compiler. In round 2, at some step the function AMStandard$reportSendDone was missed, so the TinyOS kernel variable AMStandard$state was not successfully set to FALSE and the program had no chance to send more data later.

By following the call chain(Figure 18), we found that task FramerM$PacketSent() is the place that indirectly call AMStandard$reportSendDone() and FramerM$PacketSent() cannot be guaranteed to be put in task queue every time because the task can hold at most 8 pending tasks. When it fails to put FramerM$PacketSent in task queue, disaster happens that there will be no data be able to be sent from XSM to stargate.

Further we can see from Figure 18 that the function __vector(20) is triggered by hardware event when the data transfer finished. But when this event happened and at the same time the main program is running an atomic statement, this event will be lost because atomic statement disables hardware interrupt.

<table>
<thead>
<tr>
<th>PointID:</th>
<th>S234</th>
<th>S235</th>
<th>S236</th>
<th>S237</th>
<th>S238</th>
<th>S240</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round 1</td>
<td>86473</td>
<td>86402</td>
<td>41000</td>
<td>40911</td>
<td>41549</td>
<td>86474</td>
</tr>
<tr>
<td>Round 2</td>
<td>4801</td>
<td>4797</td>
<td>2276</td>
<td>2270</td>
<td>2313</td>
<td>4801</td>
</tr>
</tbody>
</table>

Table 3 Total number of each program ID for Node N105.
static inline void TOSH_run_task(int start_point_tracker1) {
    // start to track function body
    sendPacketToStargate(start_point_tracker1, 239);
    while (TOSH_run_next_task(start_point_tracker1)) {
        // start to track while statement
        sendPacketToStargate(start_point_tracker1, 240);
    } // end to track while statement
    TOSH_sleep();
    TOSH_wait();
} // end to track function body

bool TOSH_run_next_task(int start_point_tracker1) {
    // start to track function body
    sendPacketToStargate(start_point_tracker1, 234);
    __nesc_atomic_t fInterruptFlags;
    uint8_t old_full;
    void (*func)(int start_point_tracker1);
    fInterruptFlags = __nesc_atomic_start();
    old_full = TOSH_sched_full;
    func = TOSH_queue[old_full].tp;
    sendPacketToStargate(start_point_tracker1, 235);
    if (func == (void *)0) {
        // start if tracking
        sendPacketToStargate(start_point_tracker1, 236);
        __nesc_atomic_end(fInterruptFlags);
        sendPacketToStargate(start_point_tracker1, 237);
        return 0;
    } //end to include if statement
    TOSH_queue[old_full].tp = (void *)0;
    TOSH_sched_full = (old_full + 1) & TOSH_TASK_BITMASK;
    __nesc_atomic_end(fInterruptFlags);
    func(start_point_tracker1);
    sendPacketToStargate(start_point_tracker1, 238);
    return 1;
} // end to track function body

Figure 16 Data Collecting Enabled C code for Serial Read/Write test program

Figure 17 AMStandard Module code that change value of internal varialble 'state'
5.2 DISTRIBUTED CHOWKIDAR TIER ONE PROGRAM

Chowkidar[5] program is designed as a health monitoring facility for Kansei testbed. There are three tier programs in this project. Tier one program is a TinyOS application which will run on XSM. When tier one program sends out an init message, the tier two and tier one program will build a spanning tree based on the condition of each node. The all nodes will send message to tier one program through the tree edge to report its status and its neighbor status.

In this thesis, the Chowkidar tier one program is used to test TinyOS kernel bugs. There are 1121 program point IDs are created by the new NesC compiler. Chowkidar program collected much more data here because it is a bigger program and is more complicated. Table 4 shows the total number of data collected during 4 rounds on XSM
node N67, N105, N173, N180, N218 and N233. Figure 17 shows the bar chart of the number of collected data.

<table>
<thead>
<tr>
<th></th>
<th>Round 1</th>
<th>Round 2</th>
<th>Round 3</th>
<th>Round 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>N67</td>
<td>2435238</td>
<td>3099564</td>
<td>2381444</td>
<td>2379299</td>
</tr>
<tr>
<td>N105</td>
<td>1927588</td>
<td>2170363</td>
<td>1909284</td>
<td>1903629</td>
</tr>
<tr>
<td>N173</td>
<td>2331238</td>
<td>2960061</td>
<td>2260596</td>
<td>2271269</td>
</tr>
<tr>
<td>N180</td>
<td>2471079</td>
<td>3147196</td>
<td>2412475</td>
<td>2429076</td>
</tr>
<tr>
<td>N218</td>
<td>2063932</td>
<td>2684149</td>
<td>2033746</td>
<td>2034214</td>
</tr>
<tr>
<td>N233</td>
<td>2436252</td>
<td>3095261</td>
<td>2359643</td>
<td>2371577</td>
</tr>
</tbody>
</table>

Table 4 Number of point ID numbers collected in Chowkidar tier one program

![Figure 19 Chowkidar tier one program point ID](image)
Bug #2: Buffer data corruption

TinyOS kernel implemented AMStandard module in AMStandard.nc file. This implementation will use the TOS_msg structure directly provided by user level function call. In HeartBeatM module of chowkidar program, the code shown in Figure 19 contains possible race condition. The sendPending variable is read without protection, it is possible that two modules will pass the condition check and have chance to run USend.send simultaneously. This will mess up TinyOS kernel modules when the kernel is sending data out but another code modified the buffer content. This kind of issue causes some “BAD_CRC” errors when the serial forwarder on Stargate reads data from serial port.

```c
if(!sendPending)
{
    atomic{
        sendPending = TRUE;
        memcpy(data.data, &packet, sizeof(Packet));
        crcPtr = (uint8_t *)&packet;
        data.crc = calCRC(crcPtr, sizeof(Packet));
        data.length = sizeof(Packet);
        ok = call USend.send(TOS_UART_ADDR, sizeof(Packet), &data);
    }
}
else
{
    atomic{
        memcpy(&pkt_queue[tail], &packet, sizeof(Packet));
        tail = (tail+1)%5;
    }
}
```

Figure 20 TOS_msg data corruption in HeartBeatM module
CHAPTER 6

RELATED WORKS

Recently there are some works\cite{30,31,32} done to debug sensor network applications and there are also some works\cite{27,28,29} done to secure sensor network applications.

6.1 DECLARATIVE TRACEPOINTS

Declarative tracepoints\cite{27} is the most similar work with the dynamic analysis tool provided in this thesis. It allow programmer to insert a group of action-associated checkpoints at run time without stopping and restarting the program on sensor nodes. The checkpoints are written in TraceSQL script langue and compiled into tracepoint engines executables. When the executables are installed, they instrument insert tracepoints to binaries by instrumenting their binaries. When program control flow reaches a tracepoint, the control flow will jump to an action handler which is carefully designed. In the action handler, the system will collect event information or variable values.

The system is very powerful. I can simulate many popular debuggers, for example NodeMD, StackGuard, Sympathy, etc. Also they carefully designed the event handlers to make them transparent to the whole system.

But there are some drawbacks to this system. Firstly, it requires the operating system should support dynamic loading code to run. Usually dynamic loading is too expensive for sensor network applications. Many platforms do not support it. Secondly, in order to
design good tracepoints, programmer should be very familiar with the running sensor network application. It is hard to design general tracepoints that can support all applications. Thirdly, it is not feasible to design tracepoints to handle the whole system. The code size for whole system will be comparable to the application itself.

6.2 SYMPATHY

Sympathy[28] is a sensor network debug tool that can detect the root-cause if a failure occurs. There is a central node in the system which accept messages from all others nodes. The other nodes in the system periodically send out message to the root node to report its status and neighborhood information. On the sensor node, there are many algorithms to detect failures, to find out the root-cause for the failure, etc. When a failure and its root-cause recognized, a report will be filed to users as a log file.

Unlike the dynamic analysis approach described in this thesis which aim to bugs in sensor network application itself, the Sympathy aim to the whole sensor network environment to find out which node or nodes are the root-cause for recognized failures.

6.3 AVRORA

In [29], the author provided the design and implementation of three kinds of code dynamic instrumentation for their Avrora microcontroller simulator : probs, watches and events. Probs are used to monitor a code points to record the program status when program control flow reaches that points; Watches are used to monitor actions, for example every time when a variable values changed the program status will be recorded; Events are used to monitor the system time related property. For example it can be used to record the system status every 30 seconds by inserting an even in the queue for 30
seconds in the future. When this event is triggered after 30 seconds, it will record the system status and then re-insert itself into the queue again. By this means the event can monitor the system periodically.

This approach can get precise system status when running on simulators. It takes care of the expense in running the instrumented codes and make them totally transparent to other modules, so the result can precisely show the actually simulator status.

Simulator is different with actual hardware. Although the behavior of hardware can be simulated, the environment is impossible to simulate. Watches are too expensive to implement on real hardware. For example, if we want to monitor the value change of a variable, we have to encapsulate the change of the variable in a function. Later any modification to that function has to go through this function. A lot of extra cycles wasted by doing things this way.

6.4 T-KERNEL

t-kernel[25] is a new wireless sensor device operating system kernel designed to perform extensive application code modification at load time. When loading a TinyOS application, the whole TinyOS program image will be rewritten. t-kernel’s naturalization process is applied when the load program image. The naturalization process makes sure the t-kernel can always get control when the application had some faulty code, even there is an infinite loop that would otherwise hang up the system.
t-kernel also implements a simple virtual memory support to rewritten code. A small block of consecutive instructions in the application is roughly defined as a code page. When the program control flow reaches a new code page, t-kernel will read the code page from flash memory and modify the code page to make it run on nodes. The cost of virtual memory is hard to predict because the event-driven characteristic of sensor network applications.

6.5 SAFE TINYOS

Safe TinyOS [26][17] provides efficient memory and type safety for TinyOS 2.0. TinyOS is written in NesC language which is a dialect of C language. TinyOS application will be compiled into C code before it is finally compiled to binary code. Memory and type safety is an important issue for C language. The Safe TinyOS system in [26] and [17] are essentially the same. It provides safety by modifying NesC compiler to insert safety checking code to catch memory or type access violation.

In [26], Safe TinyOS requires programmer to add deputy annotations in the programmed TinyOS application. After running modified NesC compiler, Deputy is used to enforce memory and type access safety. The newer version of Safe TinyOS in [26] used CCured to do source to source transformation that input an annotated C code and output a modified version of the code to enforce safety.

The approach implies executing additional safety checking code for every memory or type access. The cost for the additional safety checking may be huge in some applications.
CHAPTER 7

CONCLUSION AND FUTURE WORK

This chapter concludes the thesis and describes possible future works can be done in this direction.

7.1 CONCLUSION

This thesis proposes a source code based dynamic analysis approach for wireless sensor network applications. The modified NesC compiler inserts program point ID collecting statement to record the program points at which program control flow changes. The compiled program will automatically record program point IDs it reaches when it runs on sensor nodes. The collected program IDs can be used to generate many important program behavior data (as mentioned by Professor Atanas Rountev), for example, it can be used to generate (1) the number of times a function is entered/exited, (2) the number of times a function call relationship is followed in the forward or backward direction, or (3) the number of times a function call chain is followed forward or backward.
This thesis also proposed a simple approach to analysis the collected data. By comparing the total number of program point ID data collected by each sensor node, some abnormal sensor nodes can be observed. The program control flow information represented by the collected program point ID can show reason of some abnormality.

7.2 FUTURE WORK

Currently the program points collection is too heavy. It is not necessary to collect position data at each program branch unless the module needs to be thoroughly studied. The collection approach will be made configurable, i.e. we can use a configure file to control at which position a data collection statement should be inserted, at which position should not.

Currently the modified NesC compiler only insert code to collect program ID number data, but only program point ID information is not sufficient in describing program behavior. Global variable values are shared by all modules in TinyOS application. Improperly protected global shared variables can cause big problem to application. Global shared variable value will be collected in future work to show the application status when program control flow reaches a specific point.

The analysis approaches in this thesis need to be improved later. Instead only considering number of point IDs for each program point, function call chain and function call relationship will be analysis later to find more bugs.

Currently the TinyOS applications are running on XSM and data are transferred from XSM to Stargate by serial port. Because the applications we selected to test also use the serial port to transfer data from XSM to Stargate, there will be lots of conflicts between
application data packet and data collection packet. Later TelosB nodes will be used which can transfer data through USB and serial port. The quality of data collected will be improved dramatically.
REFERENCES


[22] http://www.cse.ohio-state.edu/~vinodkri/Kansei/SerialReadWriteOnXsm/


