CHALMERS

Finding Bugs in Wireless Sensor Networks with Symbolic Execution

Master of Science Thesis in the Network and Distributed Systems

Mahdieh Sadat Mirhashemi

Chalmers University of Technology
Department of Computer Science and Engineering
Goteborg, Sweden, January 2013

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.

The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company); acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Finding Bugs in Wireless Sensor Networks with Symbolic Execution Tool

MAHDIEH SADAT MIRHASHEMI

© MAHDIEH SADAT MIRHASHEMI, December 2012.

Examiner: Olaf Landsiedel

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Goteborg, Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Goteborg, Sweden, December 2012

Abstract

Sensor networks are composed of many sensor nodes which have limitations such as memory and CPU. It is difficult to access them physically, since they are usually distributed in a hard-to-reach environment such as high mountains and far deserts. They are also connected to each other by unreliable wireless links. These factors make debugging of sensor network, a challenging activity. As a matter of fact, it may be common for a deployed sensor network to encounter periodic faults. Locating and fixing these bugs are difficult after deployment. A main challenge is to find bugs that are related to non-deterministic events, such as node failure and packet loss. Since these non-deterministic events often cause complex bugs and errors in sensor applications. Most of available debugging approaches and tools have been unsuccessful in detecting such bugs.

In sensor network applications, a high coverage testing before deployment is vital to clean bugs. However, there are limited numbers of such tools available; hence comprehensive testing has been difficult so far. In this thesis work, we integrate KLEE symbolic execution tool with TinyOS to provide high coverage and effective testing before deployment. KLEE executes sensor network applications on a wide variety of symbolic inputs. In addition, by using such symbolic execution tool, we can automatically inject non-deterministic failures to applications. Consequently, KLEE highly covers application and provides numerous distributed execution paths. These paths may include low-probability situations.

As a case study, we integrate KLEE into Distance Information Protocol (DIP) and TinyOS MANET On demand (TYMO) protocols available in TinyOS package. We show that KLEE effectively detects two out-of-bound bugs in DIP protocol and three serious bugs in TYMO. Bugs in TYMO are critical and result in packet transmission failure.

Keywords: Wireless Sensor Network, Symbolic Execution Tool, KLEE, TinyOS, Dissemination, DIP, TYMO Routing Protocol.

Acknowledgment

I would like to thank my supervisor and examiner Professor Olaf Landsiedel, for his continuous support during my research and writing. I am also grateful of his patience, motivation, enthusiasm, and immense knowledge. I never thought of better one.

I would like to thank my parents for giving birth to me at the first place and supporting me spiritually throughout my life, for sympathizing me while grief surrounded me, and for being with me, when homesick, even from thousands of miles away.

Last but not least, I would like to appreciate my best and steadfast compeer, Yazdan, for his belief in me long after I'd lost it in myself, for sharing my wish, happiness, sorrow and for all his supports.

Table of Contents

1 Introduction 11
1.1 Problem statement 11
1.2 Method 12
1.3 Scientific Contribution 12
1.4 Outline 12
2 Background 13
2.1 Wireless Sensor Network 13
2.2 TinyOS 15
2.3 KLEE 16
2.4 Dissemination Protocol 21
2.5 Distance Information Protocol 22
2.6 TYMO Protocol 23
3 Related works 25
3.1 Compilers 25
3.2 Simulator and Emulator 25
3.3 Tracing tools 26
3.4 Modeling method 26
3.5 Symbolic execution tool 26
4 Design 27
4.1 KLEE Overview 27
4.2 Symbolic input 27
4.3 Network and Node modeling 29
4.4 Assertion 30
4.5 Integration to TinyOS 32
5 Implementation and Evaluation 35
5.1 Overview of implementation and evaluation phase 35
5.2 Integration into TinyOS 36
5.3 Case 1: Dissemination Protocol 37
5.4 Case 2: TYMO protocol 40

6 Conclusion 47
6.1 Limitations 47
6.2 Future work 48
References.. 49

List of Figures

FIGURE 1: MICAZ MOTE. 13
FIGURE 2: A WSN. BLUE CELLS ARE ROUTERS AND WHITE CELLS ARE NON-ROUTER NODES 14
FIGURE 3: FINITE STATE MACHINE OF TINYOS INTERFACE FOR SENDING A PACKET. 16
FIGURE 4: BINARY FILE EXPLORATION 19
FIGURE 5: KLEE EXECUTION PATH IN SENSOR ENVIRONMENT. 20
FIGURE 6: KLEE OVERVIEW. 28
FIGURE 7: KLEE BUILDS PROCESS. 31
FIGURE 8: ACT OF THE SYSTEM TOWARD RECEIVING A SYMBOLIC PACKET, B BRACHES INTO FOUR PATHS. 33
FIGURE 9: EXECUTION PATH CONTINUES IN THE OTHER NODES. 34
FIGURE 10: OVERVIEW OF INTEGRATION, NETWORK AND NODE MODEL. 35
FIGURE 11: SYMMETRIC NETWORK TOPOLOGY CONSISTING FIVE DIRECTLY CONNECTED NODES. 37
FIGURE 12: COMPONENT OF TYMO [30]. 41
FIGURE 13: SYMMETRIC NETWORK TOPOLOGY 41
FIGURE 14: ASYMMETRIC TOPOLOGY OF NETWORK (IMPLEMENTED BY KLEE). NODE 0 IS ORIGIN IN BLACK ROUTES AND TARGET IN RED ROUTES. 42
FIGURE 15: FORWARDINGENGINEM MODULE [13]. 44
FIGURE 16: BUG IN AMSEND. 45

List of Tables

TABLE 1: A SIMPLE CODE TO SHOW HOW TO USE KLEE. 18
TABLE 2: KLEE FUNCTION IN MAIN() 18
TABLE 3: LIST OF GENERATED FILES OF KLEE [8]. 19
TABLE 4: SUMMERY OF DISSEMINATION PROTOCOLS [10]. 22
TABLE 5: DIPSUMMARYP.NC COMPONENT FOR NEGOTIATING OF THE LENGTH OF THE HASH. 38
TABLE 6: PART OF THE CODE OF DIPSUMMARY FOR BUG#2. 39
TABLE 7: DIPSUMMARYP CODE. 39
TABLE 8: KLEE COVERAGE STATISTICS IN DISSEMINATION PROTOCOL. 40
TABLE 9: SENDDONE EVENT 46
TABLE 10: KLEE COVERAGE STATISTICS IN ONE EXECUTION IN TYMO 46

Chapter 1

Introduction

A Wireless Sensor Network (WSN) is a real-time, distributed network composed of sensor nodes. These nodes have an ability of monitoring and controlling physical or environmental settings such as temperature and sound. In the development of sensor applications, testing phase plays an important role. Testing software includes two main phases of finding and fixing bugs as soon as possible before deploying it into a real environment. These phases improve operation and reliability of network.

Nowadays, impediments such as memory and CPU resource limitation in sensor nodes impelled developers to use low-level languages. One drawback of low-level languages is that they have unsafe types to implement functions. Memory constraints also result in not having a dynamic type checking which prevents memory-related bug issues. Moreover, wireless sensor network has a distributed and faulty nature; therefore some bugs may exist in a program and are detected only after the software is deployed.

As an example, GSM nodes have been recently deployed in Swiss Alps. Some sink nodes were implemented for collecting measurements from the environment and sending them to a base station server. All nodes encountered huge packet loss at the same time, and the reason was a bug in GPRS driver. The bug was not detected in the testing phase before deployment [1]. Another example is a three-day network outage in the sensor network around a volcano in Ecuador, due to a bug in the flash driver of sensor nodes [2]. The bug prevented nodes to rejoin the network after a remote programming by administrators. These two examples and other bugs [3, 4] show that there is a need to comprehensively debug applications before deployment.

1.1 Problem statement

Detection of functional bugs is difficult by just checking, testing and compiling a code. Hence to detect them, one should execute the code. Bugs caused by non-deterministic events such as node reboot, packet duplicates or Packet loss, for example, are necessary to be found and fixed before deployment of WSN. The reason is that they drive the network into an undesirable state. They may cause network outage or tremendous packet loss, as an instance.

In this thesis work, we integrate KLEE, a symbolic execution tool into TinyOS, an operating system for WSN, to debug protocols. Symbolic execution tools execute a code on symbolic input

which can be anything [6]. These tools substitute an input with symbolic value and further on substitute operations with ones using those symbolic values. Consequently, we study the behavior of an application against symbolic values to inspect codes and paths.

One aim of this thesis work is to provide symbolic non-deterministic events in WSN environment and evaluate protocols and applications. We want to study the behavior of protocols toward non-deterministic event of node failure and find all bugs.

1.2 Method

In this thesis, we provide general study of WSN with a focus on routing, disseminating and KLEE as a symbolic execution tool. We also present almost all related works regarding testing and debugging WSN applications. Although some of them are far from the method used in this thesis, they are chosen to give insight into all testing approaches. To implement a testing environment, we select Dissemination and TinyOS MANET On demand (TYMO) protocols from TinyOS. The reason is that TYMO is a well-known algorithm for routing and dissemination is a basis for distributing sensor data into the network.

In addition to study these protocols and their counterparts in TinyOS, we build KLEE on top of TinyOS protocol stack to debug and evaluate these protocols. The main aim is to generate high coverage path and analyze each path and state of a sensor network while system is running in those paths. For accomplishing this goal, we model node behavior and interactions in network to provide comprehensive testing environment. Moreover, with assertion check available in KLEE library, we check sensor network in a state that is of interest for us.

1.3 Scientific Contribution

To our knowledge, integration of KLEE into wireless environment to debug protocols has not been done before. However, similar bug finding tools have been tested. We introduce them in chapter 3. This thesis work is based on two protocols selected from TinyOS package. Both of these protocols consider routing aspects of a network, since routing is almost a part of every communication in a sensor network. Hence, their reliability and security plays an important role in a packet transmission.

1.4 Outline

Chapter 2 provides a snapshot overview of WSN and TinyOS. Furthermore, we briefly describe method and protocols. Chapter 3 addresses tools that have been used nowadays in order to debug and troubleshoot wireless sensor applications. Chapter 4 focuses on the design phase, on how to successfully build KLEE on top of TinyOS and how to define libraries in KLEE. We also introduce some common non-deterministic events. Chapter 5 focuses more on experiment and evaluation phase of the project. We successfully integrate KLEE into TinyOS and detect some bugs that have not been detected in the previous testing and compilation phase.

Chapter

2

Background

In this chapter, we give some explanations about the basic concepts, purposes and an outline of this thesis. It starts from introduction into wireless sensor network and an operating system for that, namely TinyOS. Then we introduce KLEE symbolic execution tool and its functionality. Finally, this chapter ends with necessary information about TYMO, dissemination protocol and DIP, a specific library of dissemination.

2.1 Wireless Sensor Network

A Wireless Sensor Network (WSN) is a real time distributed network composed of distributed sensors. These nodes have an ability of monitoring and controlling any kind of physical or environmental settings such as temperature, sound and motion and then forward these data to a target. The number of sensor nodes in an environment varies from less than ten to a thousand depending on the size of a network. Those self-organized sensor nodes are small and low-cost with a processor (CPU, DSP or other microcontrollers) and RF transceiver. They use batteries or solar cells to provide power. MicaZ, shown in Figure 1, is a common node used in sensor network.

Figure 1: A MicaZ node.

Wireless networks are broadly used nowadays. They provide cheap and application-oriented solution for military services or disaster response situations where the other networking approaches are not feasible. The reason for using sensor networks lies behind an unlimited

potential to provide functionality in numerous applications such as military, medicine, transportation, entertainment, home security and smart spaces. Compared to a wired network, each of the existing nodes in a developed WSN acts as a router. Figure 2 shows a very simple topology of nodes functioning as a router and a non-router or intermediate. Nodes sense target data from an environment and gather all necessary events and then pass them toward the network to a node or a base station. These base stations process the information and utilize routing protocols to define outlines and rules of the data flow.

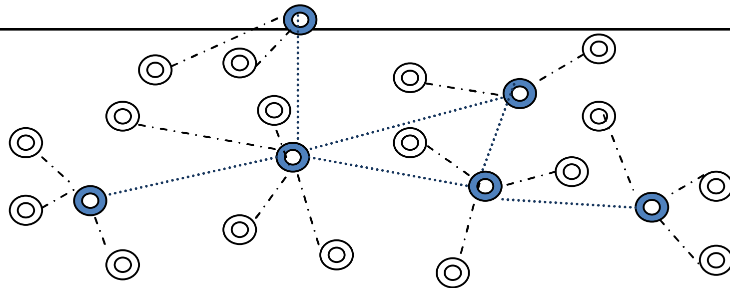

Figure 2: a WSN. Blue cells are routers and white cells are non-router nodes

2.1.1 Security challenge

As we stated earlier, sensor network nodes have limited resources of CPU and RAM which may produce faults. In addition, all transmissions are broadcast in a sensor environment and messages must be securely transmitted. Moreover, the nature of a wireless sensor network is distributed and based on interaction of nodes. It is necessary to observe interactions between nodes, although it is difficult due to complexity. Therefore, developers need to consider security, reliability and operation of WSN. It is highly recommended to test and debug wireless sensor network codes before deployment to prevent network outage.

Some critical bugs occur as a result of non-deterministic events such as packet duplicates, node crash and node reboots [5]. Existing tools for testing and debugging are not successful in detecting such events. Therefore, there is a need to implement and facilitate a testing tool integrated into WSN environment. Later in this chapter, we introduce KLEE as a symbolic tool, to address this demand.

As we mentioned earlier, each node acts as a router in a sensor network to securely deliver data to a destination. This is a main reason to investigate and evaluate routing and disseminating protocols in this thesis work. We select Distance Information Protocol (DIP) and TinyOS Dynamic MANET On-demand (TYMO) protocols as a nomination of routing and broadcast protocols.

2.2 TinyOS

Similar to cell phones, laptops and computer systems, wireless sensor networks require an operating system to execute codes and provide functionality. TinyOS is a well-known embedded operating system for WSN. It is designed to run on sensor nodes and other embedded devices of sensor network. Developers use Network Embedded Systems C (NesC) to make applications for TinyOS platform. NesC is an event-driven programming language with components wired together to execute applications [30]. An abstraction in TinyOS is designed for ultra-low power sensor microcontrollers of nodes which have limitation in RAM. Moreover, nodes' hardware has not enough support for memory safety and protection. This enforces additional testing tools and memory safe execution to ensure security in TinyOS.

2.2.1 Hardware

TinyOS executes in 8 Mega hertz on 16 bit microcontrollers. These microcontrollers typically have 4 to 10 KB of SRAM and almost 40 to 128 KB of flash memory. Programs in TinyOS are made up of software components which are connected by interfaces to each other. It is important to mention that these software components are representative of hardware abstraction as well. In order to have successful packet transmitting, routing or sensing, there are interfaces and built-in abstractions in TinyOS [12]. In the implementation phase, we build KLEE testing tool on top of these interfaces to find non-deterministic bugs in applications.

2.2.2 Abstract

The components and interfaces in TinyOS are working similar to a Finite state Machine (FSM). It means that a call of a component provides state changes; one call subsequently, calls another. These changes in states are occurred internally in components. However, some of those state changes have affection on other components. Hence components can have access to each other [14].

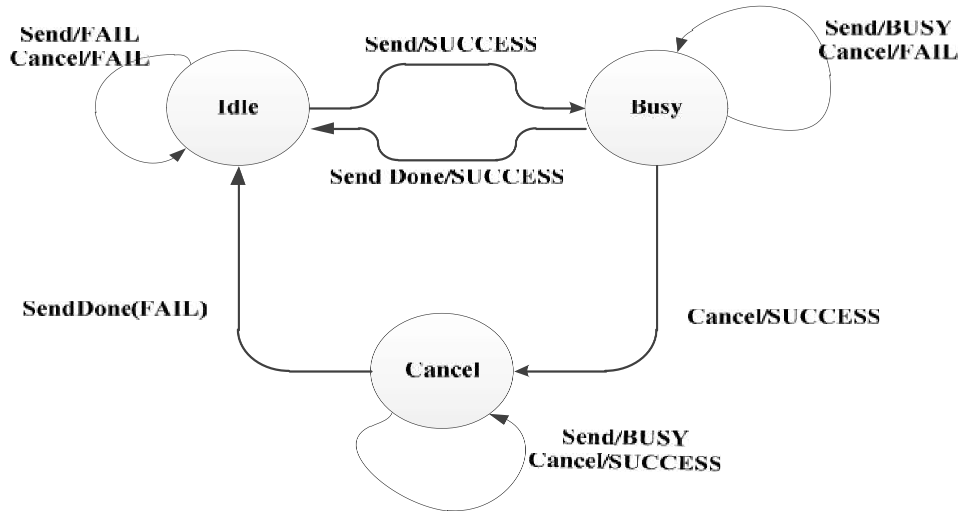

Figure 3: Finite State Machine of TinyOS interface for sending a packet.

As a very simple example, Figure 3 shows a finite state machine of TinyOS interface to send a packet. *Send* is called and it returns *SUCCESS* which causes the interface to change to *Busy* state. In the *Busy* state all other callings return *FAIL*. When *SendDone* occurs, the interface signals back with *SUCCESS* to indicate that a packet has been successfully transmitted. Then it goes to an *Idle* mode. A component at any time can call *send ()* and *cancel ().* At a point of time one call produces different results. As an instance, if a component calls *cancel ()* when FSM is busy, two options occur. Packet transmission continues which returns *FAIL* or packet transmission fails which returns *SUCCESS* and finally both return *cancel* [14].

2.3 KLEE

As stated earlier, functional bugs are a class of faults that are found when a code is executing. In addition, we should find bugs resulted from non-deterministic events, before deployment. Manual testing is difficult and does not detect all bugs. It decreases performance, while some critical errors may exist in code. Therefore, the need of a symbolic execution tool is vital. It generates an output of a program automatically while inputs are symbolic and can be everything a tester defines [5].

KLEE is an open-source, symbolic virtual machine. It is built on LLVM compiler infrastructure and acts as a symbolic execution tool to generate high coverage tests automatically [6]. Moreover, it is an optimized version of previous symbolic execution tool [7] with higher coverage. Another advantage of this tool is that KLEE's method is simpler compared to similar tools and it has more optimized constraint solving. Since it utilizes search heuristics. In addition,

KLEE enhances dealing with external environments. All aforementioned characteristics, categorize it as a high-performance and a high-coverage tool [7].

2.3.1 KLEE mechanism

KLEE considers all possible symbolic data and input while executing WSN applications. It automatically injects non-deterministic events to study reaction of an application. Moreover, it provides memory safety for TinyOS based applications [6]. In this thesis work, we apply KLEE on top of TinyOS applications. Upon hunting a bug or hitting a suspicious path, KLEE generates some log files. These files carry inputs with which the application encounters error. By investigating log files, we produce a list of faulty inputs and also we analyze them to find roots of bugs.

A main objective of KLEE is to generate high coverage path. Hence, while system is executing, we can test states of a sensor network. Below is a list of KLEE mechanism:

1. As a symbolic execution tool, it provides symbolic inputs such as a symbolic packet or sensor data to test the behavior of an application. Hence it increases the reliability aspect of security.
2. It models a network by providing a snapshot of interaction of nodes and injection of non-deterministic events. Moreover, it also models node characteristics such as node failure or node reboot. This behavior of nodes happens rarely. But if occurs, it takes distributed systems into an invalid path or state. Sometimes it causes network operation failure.
3. KLEE provides comprehensive testing environment.
4. A main check in KLEE is assertion that let us check a sensor network in a state that we aim to study behavior of the system there.

KLEE developers applied it on 89 programs in COREUTILS and consequently KLEE achieved high coverage of 90% per tool [6]. It is higher than test suits written by programmers. In addition, testers applied it on BUSYBOX and it led to coverage of 100% of 31 programs out of 75 [6]. In this thesis work, we apply KLEE on TinyOS and we have an intention to get the same high coverage.

2.3.1 Example of how to use KLEE

In this section, we show how KLEE works on a part of a code and how it finds an output according to injected inputs. Table 1 shows part of an application that gets an input and checks it according to *if* command.

Table 1: A simple code to show how to use KLEE.

```
int get_sign(int x) {
   if (x == 0)
       return 0;

  if (x < 0)
      return -1;
  else
     return 1;
}
```

To test this code with KLEE, we should execute it with symbolic integer value. So we use *klee_make_symbolic* function just after input definition in the program:

klee_make_symbolic (memory location, size, "name")

This function gets three arguments: a memory location of the value, a size and finally an arbitrary name. Table 2 shows simple function to make a variable symbolic and forward it to the program.

Table 2: KLEE function in main()

```
int main() {
      int a;
      klee_make_symbolic(&a, sizeof(a), "a");
      return get_sign(a);
  }
```

First, we compile our code and produce a bitcode *(*.bc* or **.o)* file. Then, KLEE runs on a bitcode file generated after compilation of the code. As it is shown in Figure 4, KLEE walks through three different execution paths. In one of those paths, symbolic value is 0, another one is less than 0 and last one is greater than 0. For each of these values, KLEE generates a test case which is readable with *ktest-tool* utility in KLEE directory. In each of them, KLEE gives an invoked argument, a number of symbolic objects, a name and size of the input. In the simple code above, KLEE tested a positive, a negative and a value equals to 0.

Figure 4 shows the log files of KLEE. Each of the test cases carries one input which is shown as "data" in the output (-2147483648 indicates a negative value).

```
test000001.ktest
  object    0: name: 'a'
  object    0: size: 4
  object    0: data: 1

test000002.ktest
  object    0: name: 'a'
  object    0: size: 4
  object    0: data: -2147483648

test000003.ktest
  object    0: name: 'a'
  object    0: size: 4
  object    0: data: 0
```

Figure 4: Binary file exploration

When KLEE terminates executing of paths, it generates some files in the output directory. These files represent error messages and all covered paths until termination signal receives. For each path, a file with a name of "*test<n>.ktest*" is generated by KLEE, where *n* is the number of execution. For instance, in our example code, KLEE generated three test files: test000001.ktest, test000002.ktest and test000003.ktest, as shown in figure 4.

If a path terminates with fault, or if it finds a bug, then format of output would be: "*test<n>.<error type>.err*".

This output contains information about the bug and location of it in the code. Table 3 shows a list of all files that KLEE generates. These files are useful later for analyzing the behavior of nodes and network.

Table 3: List of generated files by KLEE [8].

Name	Description
Warning.txt	Containing all warning emitted by KLEE while executing the path.
Messages.txt	Other messages rather than warnings emitted by KLEE
Assembly.ll	A version of readable LLVM bitcodes.
Run.stats	By using KLEE-stats the containing context of this file is readable. It has all statistics regarding, paths and bugs found, the times spent on the test.
Run.istats	Global statistics for each line of the code emitted by KLEE in a binary form.

Figure 5 depicts how nodes interact in sensor network. In this thesis, KLEE symbolically defines these interactions. Node A starts sending a packet to B and though initiates an execution path. There are four possible ways in node B to act toward a received packet. These four execution paths are:

1. Deliver the packet to a next neighbor (forwarding the packet)
2. Drop the packet due to wrong format or invalidity.
3. Reply back to the sender (in this case A) - to ACK receipt of the packet.
4. Deliver the packet to a neighbor and reboot.

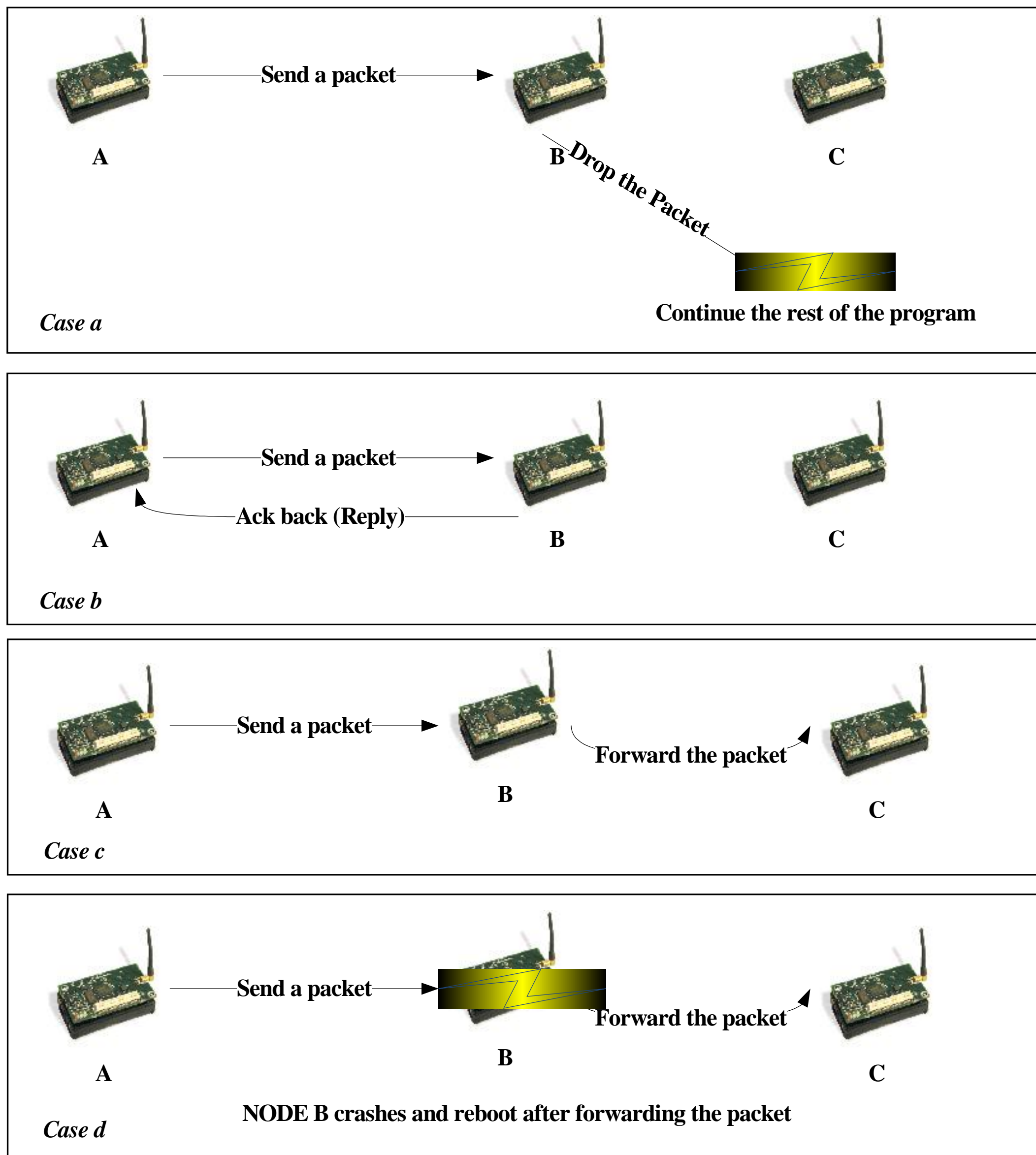

Figure 5: KLEE execution paths in sensor environment.

Consequently, node c has same execution paths after receiving a packet from B. Moreover, in other paths, program executes in normal way until reaches end of the program or faces an error. As it appears, KLEE covers all possible reactions of node B upon receiving a packet. Figure 5 shows non-deterministic events such as node reboot (depicted in Figure 5d) or packet loss (depicted in Figure 5b).

We consider above topology in Chapters 4 and 5, with a greater dimension for testing protocols, and integrating KLEE into TinyOS. More information about KLEE is also available at [6, 8].

2.4 Dissemination Protocol

In wireless sensor network, dissemination is a service to broadcast a small packet in network. Receiving nodes do not acknowledge back when they get a message. In sensor network, every node is keeping a copy of a variable whose value is shared between all nodes (such as temperature, vibration or emission). The ultimate goal of this protocol is to provide consistency on shared variables [9].

2.4.1 Consistency

Dissemination protocol forces node to disseminate a value whenever a change in that value is detected. Therefore, eventually all nodes across network have the same value as the disseminator and they are consistent. It may happen that some nodes do not agree on a new value, but after a couple of minutes network converges to that new broadcast value. Dissemination protocol promises that over time, network is consistent, if disconnection does not happen. However, consistency is threatened by packet loss and disconnections which happens temporarily in networks. A reason may be due to mobility of nodes. These threats, force considering security evaluation on this protocol and motivated us to apply KLEE on it to completely test.

Consistency does not refer to that all nodes receive every new value for a variable and substitute an existing value. In this case, bandwidth is saturated by un-necessary and extra overheads. Consistency indicates that all nodes eventually agree on the most recent value that a variable gets and apply that to what they stored. Then, if several nodes have a same opinion of updating their value to the most recent version, dissemination protocol concludes that the network is converged to a single value.

Overall, usefulness of dissemination is vivid in a situation that a network administrator would like to inject a special command, script or any constants into an application. This remote administration is for a purpose of upgrading or installing a small program into a network to build an application based on sensor network.

2.4.2 Dissemination libraries

In order to provide eventual consistency model, there exists three dissemination libraries in TinyOS namely Drip, DIP and DHV. A basic difference between these protocols is that in Drip, each data is considered to be a separate entity for dissemination. Hence there is a control over time of dissemination and speed in which data is disseminated. It is useful when a small number

of data is available and there is no information about where those data are stored. However, in DIP and DHV data are treated as a group, so dissemination control applies to all. In this case there is an agreement on the data and less messaging occurs in network, results in bandwidth saving [10]. Totally, in DHV fewer messages are exchanged between nodes. So, DHV is faster than DIP and DRIP. Table 4 shows differences in greater detail.

Table 4: Summery of Dissemination protocols [10].

	Radio control	Timing	Messaging	Data size
DRIP	Application is responsible of enabling or disabling radio for dissemination	**DisseminationTimerP** component trickle timers. The tau values apply to all dissemination items.	Each data item advertised and disseminated independently. Metadata is not shared among data. So nodes do not need to agree on data sets.	Typedef **t** controls data size. It should be smaller than the message payload size.
DIP	Application is responsible of enabling or disabling radio for dissemination	A single trickle timer is for all data items. Tau value is configured in **Dip.h**	For all fixed data set an advertisement message is used and all nodes are to agree on a fixed data before dissemination. DIP advertisement is based on message size in **Dip.h**	Typedef **t** controls data size but nx-struct **dip-data-t** in **Dip.h** is a backup. The size in less than the message payload size.
DHV	Automatically start the radio. And manually turning off the radio which causes DHV not to work through.	A single trickle timer is for all data items. Tau value is configured in **Dhv.h**	For all fixed data set an advertisement message is used and all nodes are to agree on a fixed data before dissemination. DIP advertisement is based on message size in **Dhv.h**	Typedef **t** controls data size but nx-struct **dhv-data-t** in **Dhv.h** is a backup. The size in less than the message payload size.

In this thesis work, we investigate DIP library, since it is a protocol in use nowadays. The next section provides a short introduction to this protocol.

2.5 Distance Information Protocol

Distance Information Protocol (DIP) is a data detection and dissemination protocol. It has two fundamental functionalities. It detects a difference in values and also identifies the time in which, that difference has occurred. DIP protocol utilizes search methodology dynamically to find changes in the value [11].

DIP continuously calculates the probability of data item by exchanging messages. When the probability reaches 100 %, data exchanging starts and lasts until all nodes eventually have consistent set of data. DIP protocol keeps a version number for each data. A version number is an indication of steady state and shows that all nodes are up-to-date. Each time, a hash containing a version number, is sent. If a node receives a hash, it checks it against its own value. If values are the same, it means that it is in a consistent state. But if they are different, the node concludes that there is a change in value, however, it has no idea of which part is different. DIP also stores the state estimate, so in the time that the estimation is incorrect, protocol continues execution.

In this protocol, it is essential to identify which data is new and what node owns a newer version of the data. This process requires exchanging version numbers among node in a hash tree basis. This hash tree has a bug in its code. We address this bug in Chapter 5.

2.6 TYMO Protocol

In WSN, message transmission plays an important role, since all communications are based on these transactions. Hence there is a need to have routing protocol in order to fulfill the need of switching information between nodes and send or receive messages.

TYMO is a routing protocol in TinyOS designed for this goal. IETF designed the first draft of TYMO called DYMO (Dynamic MANET on Demand). DYMO is a successor of Ad hoc On-Demand Distance Vector[1] (AODV) protocol. The main goal is to dynamically find routes over IP stack. TinyOS implemented a version of DYMO protocol as a routing protocol for point-to-point purposes, applicable in Mobile Ad-hoc networks (MANETs) with a slight change in the protocol. The packet format differs from DYMO and the implementation is based on Active Message stack. It is designed in order to make the network layer modular [13].

As discussed earlier in this chapter, one of the limitations of sensor nodes is lack of memory. TYMO is applicable for memory constraint sensor nodes, since in TYMO, network topology is not stored in nodes. The only information stored in a node is a little routing information which is exchanged in the network when needed.

2.6.1 Concept of TYMO routes

TYMO has its own way of handling routes. When a node wants a route to a destination, it disseminates a packet to get a route between originator and destination. This small packet is called Route Request (*RREQ*) which is disseminated in the network within a defined number of hops. Hence, network traffic overhead reduces. Furthermore, bandwidth and power of nodes are saved which is desirable for sensor nodes. When packet arrives at the destination or at a node that has enough information about the target node, a Route Reply (*RREP*) packet is sent back to

---

[1] It is a routing protocol for ad-hoc mobile networks. It is designed for both unicast and multicast routing. When source demands, routes between nodes are built and are kept as long as they are needed by the sources.

the originator. Format of both *RREP* and *RREQ* messages are similar to each other. A difference is that *RREP* is unicast and no acknowledgment is sent back [13].

In *RREQ* and *RREP*, information about both sender and originator of the packet are stored in the cache of every passing node. Hence, nodes can use this information in future packet exchange. When a node receives a packet, it collects paths which have been followed by the packet. These accumulations would be useful for another packet that arrives at a node, in order to calculate a path to target.

TYMO uses sequence number and hop count, similar to most of other routing protocols, in order to check quality of routes and prevent loops in the network. Moreover, in order to have fresh information in a small routing table inside nodes, data and information which has not been used for a long time is deleted regularly by nodes. If a packet arrives and needs to access information which has been recently deleted, a Route Error (*RERR*) massage is generated. This message tells other nodes that the route does not exist any longer [13].

In order to implement TYMO, a transport protocol implemented with the responsibility of forwarding a packet to the target. This protocol is called MH (Multi Hop) and uses relevant interfaces of TinyOS such as *AMPacket*, *splitcontrol*, *AMsend*, *receive* and *packet*. Later in Chapter 5, we evaluate MH protocol and other dependent interfaces. we show that, this Protocol has serious bugs needed to be fixed.

Chapter

3

Related works

Generally, a majority of evaluations and tests in wireless sensor networks is based on using a combination of tools. There exist lots of approaches and tools to discover bugs. Some of them focus on debugging before deployment while others are for testing after a code is deployed. In this section, we classify and explore testing tools and approaches in WSN. Although some of these tools are not based on symbolic execution which is an aim of this thesis, they show the significance of providing security in sensor networks. Moreover, a combination of these tools provides better security for a system.

3.1 Compilers

A first step of testing a code is Compiling. We primarily use compilers to perform lexical and semantic analysis, parsing, code generation and optimization. Compilers detect program faults related to errors such as out-of-bound memory, wrong type conversions and race conditions. A good compiler in WSN is safe TinyOS, a tool to apply type and memory safe application at run time [12]. Safe TinyOS detects most of harmful memory errors that cause OS code and RAM corruption in nodes. This compiler is for TinyOS environment.

Neutron [24] is another similar tool, to enforce memory safety and bug recovery. When a code is deployed, it would be difficult to debug it. In addition, events such as battery usage, concurrency errors and failure in connectivity of nodes exist in sensor networks. Hence, they make it hard to diagnose which error is caused by memory issue and which is caused by other events. Overall, these tools are useful to detect local errors in an implementation. However, there are a wide variety of distributed bugs caused by interaction of nodes, which may remain unsolved even after compiling..

3.2 Simulator and Emulator

Simulators and emulators such as TOSSIM [16], COOJA [18] and ATEMU [19] provide solutions to test and debug network. Stackguard [28] is a tool to detect buffer over-run security exploit, which causes stack corruption. Sensor network troubleshooting suite (SNTS) [29] is another tool to check packet format in order to detect problems such as incorrect sequences of event. It uses some nodes as a sniffer to provide an overview of network traffic. Then they transfer the information to a central server for further processing. Another tool is ANDES [15] which is a tool to detect the root of anomaly operation in sensor networks. One of the most

important limitations of these methods is low-input coverage. Hence, they do not provide low-probable scenarios in which inputs cause bugs. Therefore, it is better to combine simulators and emulators with other methods in order to close this gap.

3.3 Tracing tools

Tracing a specific part of a code is of importance, since it detects significant errors in that part of the code. Approaches like TraceSQL [20] and NodeMD [21] are designed for this purpose. Another tool is Marionette [21] that provides capability to trace at runtime. There are also some other approaches such as PDA [22], EnviroLOG [17] and MDB [23]. They are designed to collect states of node and then according to the collected data, analyze the network. But a limitation of all these and similar tools is that they are solely capable of detecting errors and bugs in a part of the execution trace. Hence they analyze a particular part of an application and do not provide a full coverage scenario.

3.4 Modeling method

Some model-oriented methods are developed in order to analyze sensor network codes. They check applications by modeling method, to assure their accuracy. Initially, these methods take a whole model of a system and then inject it into a specific environment to verify the correctness. In [25-27] some of these methods are discussed.

Overall, there are some limitations and complexities in these methods. First of all, they require hard endeavor to provide a valid model for each system. Secondly, they are not suitable to detect explosion in states. This error occurs as a result of non-deterministic nature of wireless sensor network.

3.5 Symbolic execution tool

Tools like KLEEnet [5], FSMGen [14] and KLEE are designed for the purpose of executing a code on a symbolic value. Their model is based on a state machine. One of the advantages of KLEE over other similar tools is that, it applies symbolic execution technique on behavior of nodes. This offers high execution path coverage and high input range. Moreover, it runs modified application and increases accuracy.

Last but not least, KLEE detects distributed interaction bugs with a possibility to model events such as packet loss, packet duplicates and node outage. As we discussed earlier, these are examples of non-deterministic events. These reasons motivated us to select KLEE and build this tool on top of TinyOS for debugging applications.

Chapter 4

Design

This thesis aims at integration of KLEE as a symbolic execution tool into TinyOS to debug protocols. In this section, we discuss how to build KLEE on top of TinyOS to automatically generate paths. Our goal is to study the behavior of sensor applications while non-deterministic events such as node failure and node outage occur. Moreover, we explore network and node modeling to address interaction of nodes and topology of network. Finally, we produce a test environment to find insidious bugs in TYMO and dissemination protocol.

4.1 KLEE Overview

As discussed in Chapter 2, there is a demand to provide high-coverage in execution of paths to detect bugs in WSN. We chose KLEE to provide automated path generation in distributed sensor network. To accomplish this goal, KLEE has a specific approach which we discuss in this chapter.

Figure 6 demonstrates a general overview of the design phase. There are four different categories in this phase. As a symbolic execution tool, KLEE provides symbolic inputs such as symbolic packet or sensor data to test the behavior of an application. When KLEE uses symbolic values, we put assertion in special parts of the code to check the application in different states. Then KLEE models network and provides snapshot of interaction of nodes. KLEE also injects non-deterministic events such as packet loss or packet duplicates into the network and transfers messages among nodes. After defining network model, it is time to model node characteristic such as node failure or reboot. This behavior of nodes happens rarely, but if occurs, it causes undesirable afterwards. In the following sections, we present a detailed explanation of the design phase.

4.2 Symbolic input

KLEE is designed to make an input variable and data as symbolic, see Chapter 2. These inputs are data gathered from an environment such as temperature, frequencies or packet transmitted between nodes. In a sensor network, the first step to walk through an executing path is to symbolically inject an input to an application. The question is that, in order to test and troubleshoot the distributed sensor network applications, what inputs should be defined as symbolic. In sensor networks, transmitting Packets and information related to sensor nodes play important roles; hence it is a desirable scenario to make those symbolic [5].

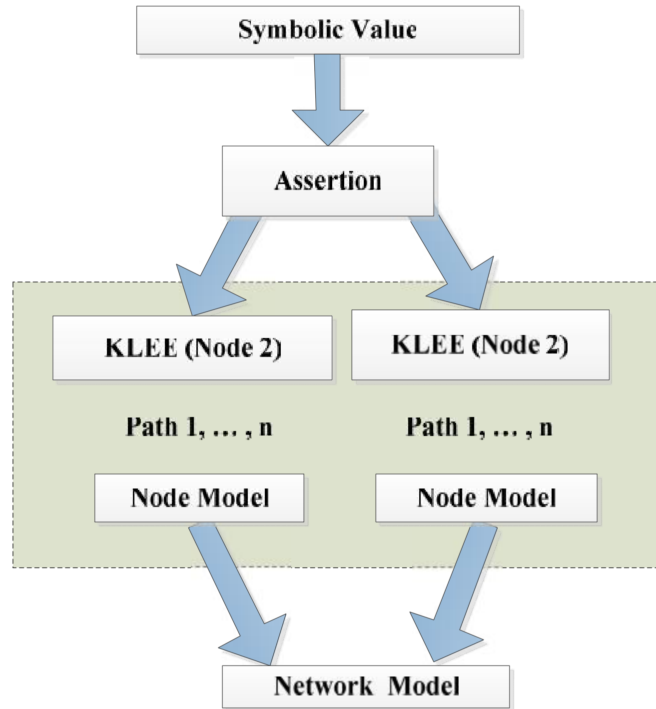

Figure 6: KLEE overview.

It is of importance to have symbolic input to test functionality and correctness of code operation in the beginning of development phase and further on the code. As an instance, developers of a new routing protocol may want to know:

1. What happens if a node in the path from source to destination fails in the middle of a transition.
2. How a protocol reacts if a path is not accessible.

Moreover, a developer might desire to have symbolic header in a packet for further study of execution paths, while disseminating a packet to all nodes. Hence, this provides feasibility to be sure that all nodes have same values at the end of execution. In addition, we know the effect of packet loss or a special data input on an application.

So, this testing assures us that at the end the state of the application is valid and a variety of inputs and sensor data does not affect the operation and functionality of the application. Developers would continue with feedbacks received from analysis to make the code bug-free.

4.2.1 Complexity

There is a complexity factor in KLEE, similar to other execution methods. It determines time spent totally on a testing or debugging scenario. The best scenario is that in the first execution path, we find a bug. The worst scenario, similar to what is calculated in [5] is$O(t^{n.m})$. Where n is the n-ary branch, m is the number of nodes and t is the time. In our implementation and evaluation phase, we observe that the complexity would not reach the worst case.

Moreover, in experimental part of this thesis, growth is not exponential. Even in a communication protocol such as TYMO, where numbers of nodes exceed hundred. So, we conclude that the execution paths do not grow exponentially. Hence, protocols under test, TYMO and dissemination, both have a small number of execution paths. In the next section, we discuss modeling of node interaction and topology of network in more details.

4.3 Network and Node modeling

As a part of evaluation and debugging of protocol stacks in TinyOS, it is necessary to model nodes characteristic such as node failure or node reboot. As we mentioned earlier, these behaviors are categorized as non-deterministic events. Another important modeling is related to network operation. It models an interaction of nodes and network-based, non-deterministic events such as packet loss or packet duplicate.

As we discussed in chapter 2, a wireless node is a hardware device which is exposure to software reboot due to memory faults, battery outage and hardware faults. Some of these problems cause a node to reset and the node may lose unsaved data or states. Although occurrence of these outages is too rare, there is still probability. So we need to wisely consider them. It is important to study and test the path in which a node fails in order to test a code. In sensor networks, nodes are neighboring. Therefore, when a node fails, neighbors by default should detect this failure. Otherwise, distributed sensor network may fail or may go to an invalid or undefined state [5].

In the following sections, we provide a short discussion of common non-deterministic events that cause problem in a sensor network.

4.3.1 Node failure

In order to introduce node failure in sensor nodes, KLEE provides symbolic event to reboot one of the nodes in the middle of an execution. After execution terminates, we observe the behavior and reaction of our system. Program executes in two valid paths. One is an execution that all nodes are working well and connected to each other. In the second path, one or more nodes have no connection to the rest of nodes in sensor network.

In node reboot or node failure scenario, we ask KLEE to disconnect a randomly selected node from the rest of network for a while. After a specific time, KLEE connects the node to one or more sensor nodes in network. Therefore, execution in that path re-initiated. KLEE records all states in a file which is useful for later analysis. We investigate all generated files and results, in order to find out whether promising goals of the application meet.

4.3.2 Packet loss or corruption

Since the operation of sensor network is based on packet transition, packet loss may be a common failure in the network. KLEE has an ability to introduce packet drops symbolically. Therefore, we can verify the effect of this event in sensor network. In the middle of an execution, KLEE discards a transmitting packet and records behavior of network toward this distributed non-deterministic event. These records provide a reference to study network operation.

By symbolically dropping an arbitrary packet in a network, KLEE introduces a possible buggy scenario and covers it. It also provides another new distributed execution path, by symbolically corrupt a packet and continue execution to see what happens to the rest of network.

4.3.3 Packet duplicates

A possible fault in sensor network is to have re-transmitted packets or to have loops. So, investigating network by injecting duplicate packets can be of interest in terms of finding faults and bugs. KLEE introduces packet duplicate symbolically to the network in order to see robustness of dissemination and TYMO in different execution phases.

Modeling non-deterministic events gives the opportunity to cover all execution paths in codes to find insidious bugs. This is a reason that we consider those events in the design phase. We addressed earlier that complexity of network increases by introducing all events simultaneously. Hence, we select events which are happening more frequent in protocols and have critical outcomes.

In the implementation phase in Chapter 5, we study the behavior of two sample protocols, DIP and TYMO. In case of symbolic input in TYMO, we observe that node failure is mainly a root cause to produce faults in network. In addition, we study the behavior of dissemination in case of having a symbolic packet with different sizes and types. We also show that rather than bugs related to symbolic input and network model, we find a few memory-related and configuration issues in both protocols.

4.4 Assertion

There may be numerous execution paths in a sensor distributed network. It is time consuming to analyze every path to get an overview of what has happened and what was the result. Although there are some tools in order to show results (such as *klee-stat*), we still need manual analysis of each of the results separately. Moreover, KLEE covers all paths and states among which some are un-necessary for us (and for each one, KLEE defines one test case). What is important for us is to explore paths in which abnormalities occur. In order to test protocols, it is more convenient to check that:

1. Nodes inside a conversion domain are in correct state, after protocol is applied.
2. If protocol is doing as it should.

For example, in a sensor environment that uses routing protocol to transfer messages, it would be interesting to know what happens to ACK messages or to entries of routing table, if one node fails. In such a case, with help of assert in ACK controller event, we study violation against a condition. This condition can be a simple statement of “if a node does not receive an ACK in 5 seconds (a defined duration)”. Consequently, KLEE generates an error file. Then, we investigate these files deeply to find, what is a cause and root of this violation. With use of assert, execution

is limited to a narrow section, hence, it reduces manual effort in finding bugs in numerous generated paths.

For more explanation of assertion, assume *f(x)* and *g(x)* implement the same interfaces. We make *x* as symbolic by KLEE. Then we execute KLEE on assert *(f(x) == g(x))*. In this case, for each explored paths, KLEE terminates with or without errors of "paths are equivalent" or "mismatch found".

KLEE proves assertions on a per path basis. It considers that all constraints do not have approximations. An assert is a kind of a branch, and KLEE shows feasibility of each branch it reaches. If KLEE determines infeasibility of false side of assert, then it proves the assert on the current path [6].

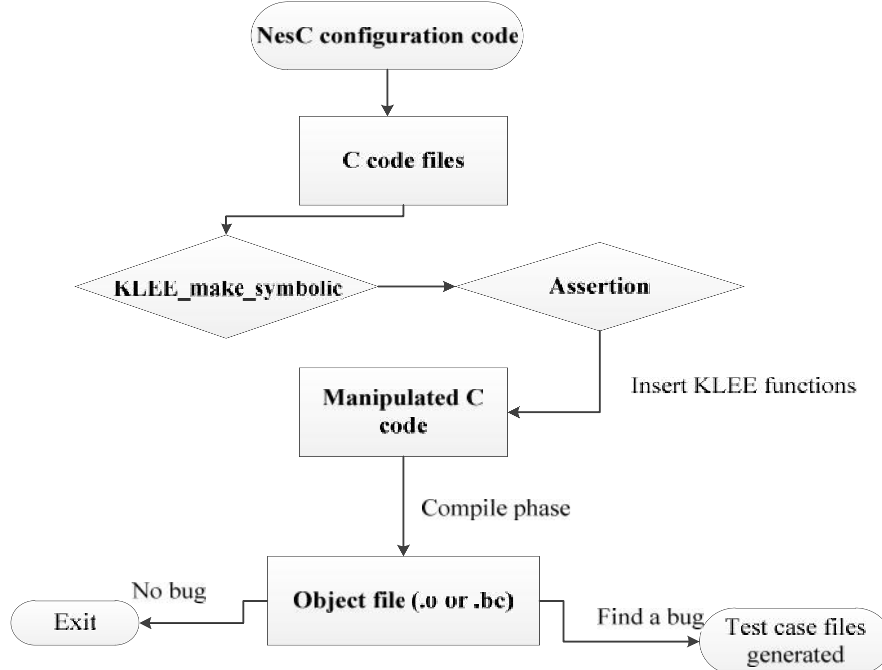

Figure 7: KLEE build process.

In order to use assertion functionality in a sensor network application, we need knowledge of data and code structure of applications. We should deeply revise the code to know where is appropriate to have beneficial assertions.

So far, we addressed considerations in design phase and in node and network modeling .The next section provides an instance of how to integrate KLEE with TinyOS.

4.5 Integration to TinyOS

As discussed earlier, we want to integrate KLEE into TinyOS by adding KLEE platform based on the OS null platform. This permits marking a sensor value input and a packet transmitting in network as symbolic. Event-driven nature of sensor applications enforces execution of some codes only after another dependent event is fired. Hence, to cover all paths of an application regardless of related events, the main code is manipulated to signal all events automatically. When the application starts, all events are fired to be processed and included in test.

4.5.1 Process of KLEE build

Figure 7, demonstrates an overview of integration process. We should take the following steps to accomplish integration:

1. We need to specify which variables should be symbolic in the application code (files contain configuration data). To succeed, we insert a call to *Klee-make-symbolic* function in the "*.c*" code (a *".c*" code is generated after parsing NesC code).
2. KLEE marks packet buffers, number of nodes or node connectivity as symbolic. These are inputs that check the state of a program regarding node failure or packet loss. At each execution, it is better to mark one event as symbolic to avoid complexity in results.
3. KLEE operates on LLVM bitcodes. In order to run a program with KLEE, we should compile the code with LLVM compiler.
4. At the last step, we run KLEE on the bitcode file to interpret it. At this step, KLEE generates test cases and writes them to files with *.ktest* extension. In case an error occurs, KLEE writes some additional information about the error into a file namely *testN.TYPE.err.*

Generated test cases indicate which input causes error and which inputs are safe. Hence, by analyzing test cases and by considering network and node model provided by KLEE, we can find faults and bugs in an application.

4.5.2 Example of a node modeling

Applications in TinyOS use non-symbolic data. Simulators such as TOSSIM, use real input in sensor applications. Therefore, limited range of data is available for testers. KLEE works similar to simulators, but injects symbolic inputs into a sensor network application. In KLEE, when an executing path encounters an error in a code with a symbolic input, that execution path is branched into a new one. Hence in order to keep an execution in a consistent level, we should observe and investigate these paths in the provided network model.

As an example of a node model, consider a network consists of 4 nodes. BY KLEE, we symbolically define a packet and inject it into a distributed system. Node A sends a packet to node B. After receiving a packet, there are some possible ways in node B. Generally, depending on an implemented protocol, B sends back an ACK after successfully receiving a packet or drops

it and sends a NACK back. There are other possibilities such as forwarding the packet to next nodes or discard without any further action.

We say there are four execution paths defined in node B. Figure 8, shows this approach with all possible execution paths.

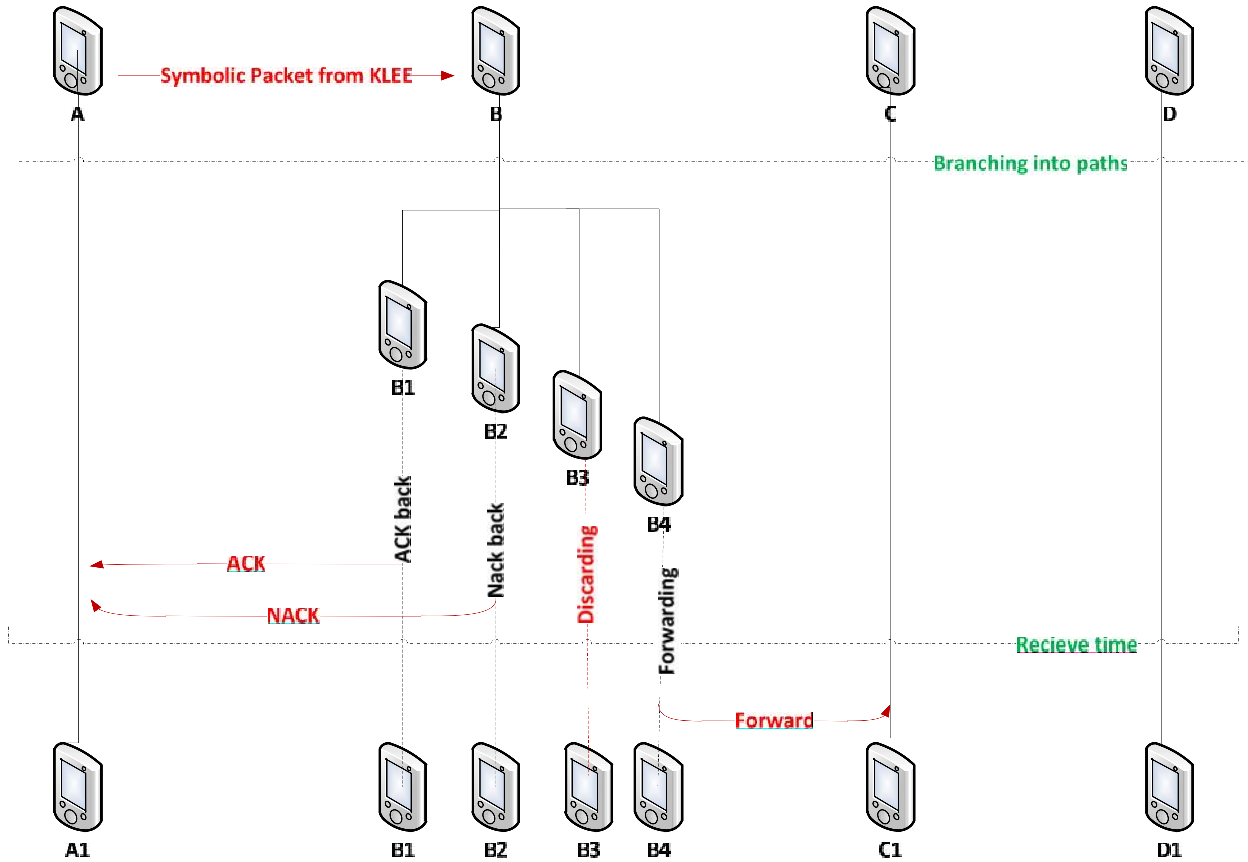

Figure 8: Act of a system toward receiving a symbolic packet, B braches into four paths.

Later in Chapter 5, we use this approach to debug TYMO Routing protocol.

Node C has a similar approach to node B. Upon receiving a packet from node B, it branches into four other execution paths as depicted in Figure 9. Many of execution paths are unnecessary and lead to waste of memory resources. It also requires a long execution time. In order to optimize the execution time, we include an assertion in the codes. It enhances paths and focus only on part of the execution desired for us. The effect of an assertion mainly is visible in a sensor network with large network setup such as a routing environment. In such a network, a lot of nodes are exchanging sensor information and a packet influences a big part of the network. So, if we limit our study on the effected part of the network, it helps us to efficiently debug the protocol.

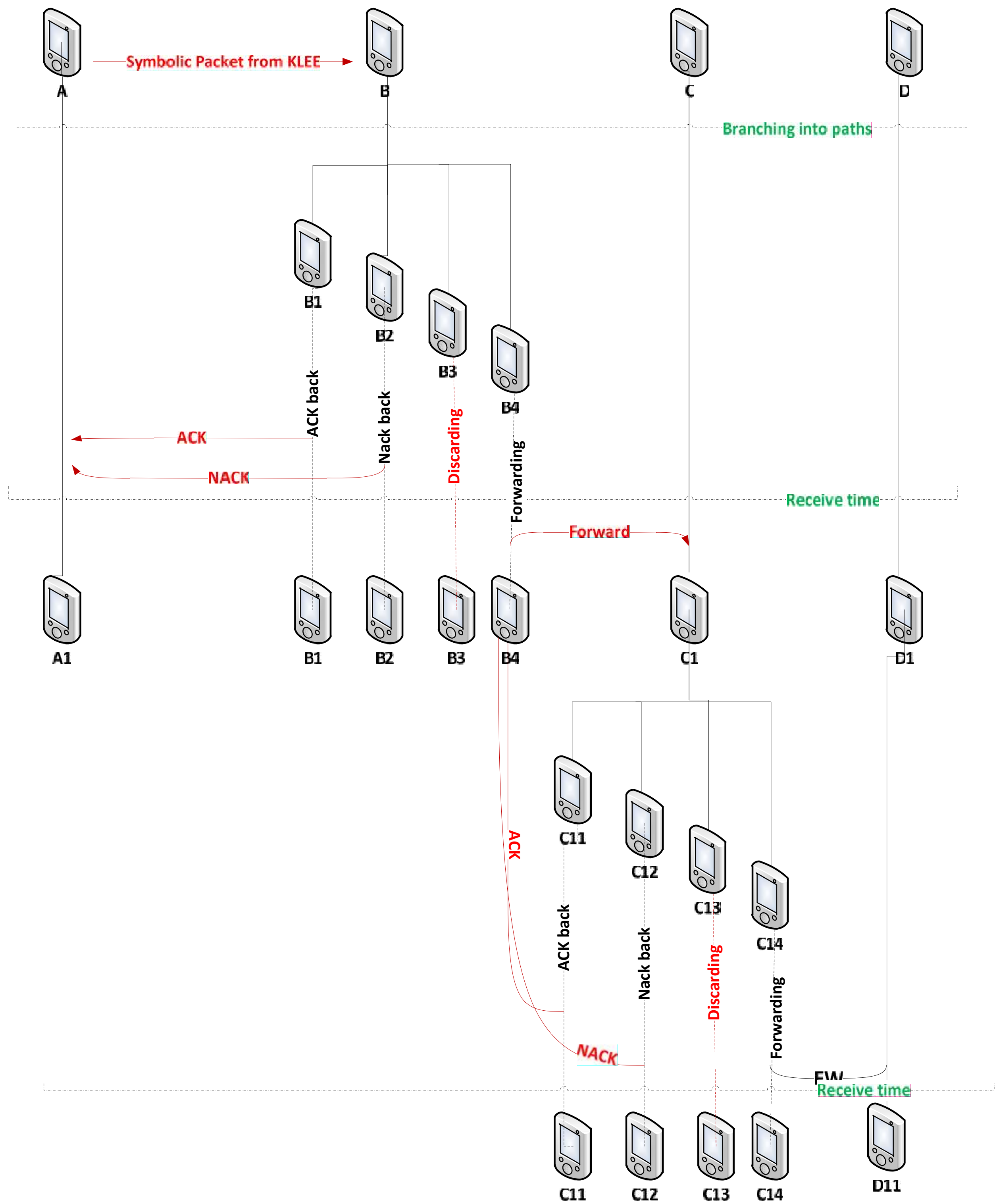

Figure 9: Execution path continues in a subsequent node.

Chapter

5

Implementation and Evaluation

In this chapter, we provide an overview of the implementation and evaluation phase of this thesis work. To begin with, we define a node and network model, as discussed in the previous section. Then we implement a scenario for testing two sample protocols. Upon successful integration of KLEE and TinyOS, we evaluate protocols in order to debug them. Results show that KLEE found two memory faults in dissemination protocol and three bugs in TYMO.

5.1 Overview of implementation and evaluation phase

Figure 10, demonstrates a general overview of the implementation and evaluation part. As it appears, we include node outage and reboot as a node model together with packet failure as a network model, to evaluate protocols. There are two phases. First, we verify a feasibility of integration into TinyOS. Second, we consider some case studies to evaluate the design. At the

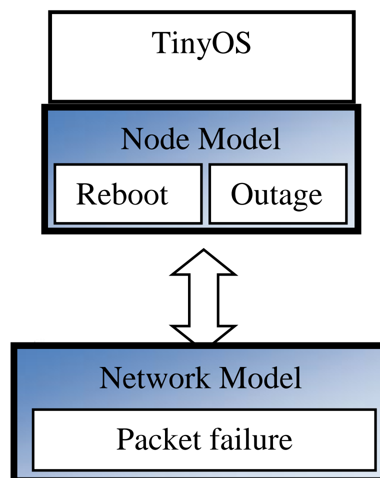

Figure 10: Overview of integration, network and node model.

first case study, we use KLEE on top of TinyOS in order to test functionality of DIP protocol. In the second case study, we run some tests on TYMO protocol and use KLEE to inject packets and alter topology of the network.

5.2 Integration into TinyOS

KLEE has a design independent of platform, which allows it to be applicable to any kind of operating system such as Contiki, Kooja or TinyOS [6]. In order to integrate KLEE into TinyOS, which is a goal of this thesis, we utilize some libraries of KLEE such as replay and assertion. This minimizes hard work of large log file analyzing, by just narrowing the execution to a single path.

Similar to simulation tools such as TOSSIM, node and network modeling in KLEE are based on hardware layer. A difference is that all events are symbolic in KLEE, but they are manually defined by user in simulation environment [5]. There is a possibility to introduce node reboot and node failure such as what may happen in real environment. Moreover, while sensor nodes communicating with each other in a sensor network, KLEE can inject events of sending packer, receiving, forwarding or even discarding. This integration is shown in figure 10.

As discussed in Chapter 4, KLEE can implement node outage at any time during execution. This consequently, produces more execution paths. Contrary to TOSSIM, packet loss is not done with Channels. Here, KLEE decides to branch an execution path into different paths in a certain time. One path is to pass the packet to a neighbor node and another is to discard it, in order to simulate packet loss in the network. Similarly, a scenario for packet duplicate is that right after receiving a packet, KLEE forks the execution path. One path may be a successful delivery of packet to a next node and another is to produce the same packet twice and inject them to network. Similar approach is applicable for packet corruption.

KLEE can control time of execution in order to prevent state explosion. This time is used in implementation phase to limit execution and to save bandwidth and CPU of processing computer. Another necessity is to model the topology as symmetric or asymmetric. In symmetric topology, every node has one connection to other nodes in network. While in asymmetric topology, nodes have arbitrary connections. There might be a node in asymmetric topology which is connected to only one node, whereas rests are connected to each other.

We also define a number of nodes and the connectivity between them in the topologies by KLEE. Hence, we study the behavior of protocols toward increasing network size. This also provides simulation of the probability of node disconnection and increased load while the application is executing in a real environment. In this case, we may want to have a complex model of network with multi-hops and many nodes. These nodes have direct connections to some other nodes or do not have any connections. However, network can be as simple as a small full mesh network.

In both aforementioned scenarios, we analyze protocols separately, to investigate all possible reactions. We also consider non-deterministic event of node failure in setting up KLEE environment. The motive behind choosing node failure is that, this event has an important effect on routing and disseminating of packet in distributed sensor networks.

5.3 Case 1: Dissemination Protocol

As discussed in chapter 2, dissemination is a service to broadcast a small packet in the network to provide consistency on shared variables between nodes. Whenever there is a change in a value, the detector node distributes the value to all nodes. Therefore, all nodes have the same value as the disseminator [10].

The first case study is to test dissemination protocol. DIP is a data discovery and dissemination protocol. It is one of the existing libraries in dissemination protocol. Topology of our network is symmetric, meaning that there is at least one path from one node to another. In a very well scenario, all nodes are up and working.

Our test starts with 4 nodes. But during execution, a number of nodes increases to provide a full-coverage scenario with as many numbers of nodes as possible (We continue until CPU and memory of computer allows). Figure 11 demonstrates a simple case of a symmetric network topology consists of 5 nodes. As it appears from the figure, all nodes are connected directly to each other.

xWe select *TestDisseminationAppC* code in TinyOS package as a source to test the functionality of dissemination protocol. Since this package comprehensively utilizes the dissemination layer. In this application, every 4 seconds, node with ID 1 injects two new values of 32-bit and 16-bit into the network. 32-bit value uses key 0x1234 and 16-bit value uses key 0x2345.

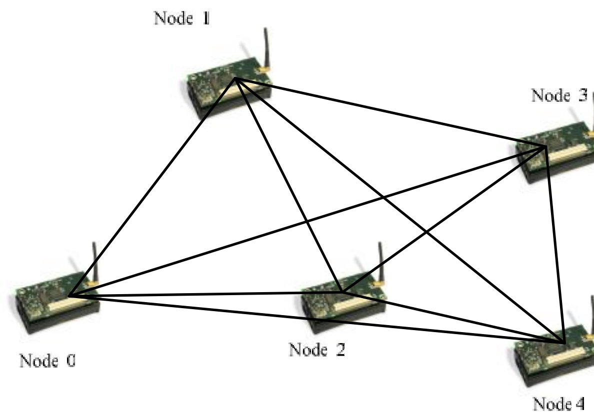

Figure 11: Symmetric network topology consisting o five directly connected nodes.

Both sender and receiver nodes are using Light-Emitting Diode (LED) to indicate operation in a real network. However, in testing process, we use *debug* and *printf* commands in the main code and related tasks. A disseminator starts the protocol by sending a packet to network. Nodes

receiving that packet signal by blinking LED and apply the value. More detailed information of tasks and functions of dissemination are available at [9].

Right after using KLEE in the program, we found two memory issues. KLEE writes them in log files accessible to us. Below we describe these errors.

5.3.1 Bug#1: Out-of-bound Pointer Error

DIP protocol bases its decision on a probability of data changed, see Chapter 2. At a certain period of time, the protocol sends hashes to nodes in order to extract version numbers. If a node receives a hash, it checks it against its own value. If they are the same, it means that the node is in consistent state. If they are different, the node concludes that there is a difference [9]. Hash functionality plays an important role in providing base of DIP decision. Therefore, it is sufficient for us to make sure that this part is securely implemented and works as perfect as it should. In order to handle hash, developers implemented *DIPSummary.nc* component. In this component, *DIPSummary* message is responsible for summarizing and hashing the extracted information from a range of data. Hence we choose this part of the code and apply KLEE on it.

Table 5 shows part of a code in *DIPSummary.nc* that carries a bug. This bug is related to a case where *getPayloadPtr()* returns null. Then, it causes *dmsg* to dereference a NULL pointer in line 13 of Table 5. This error is categorized as a safety violation.

Bug#1 is so-called as out-of-bound error. At the first compilation phase, LLVM compiler did not find this bug, while KLEE successfully found it. This bug shows importance of having powerful and full-coverage testing tool of KLEE type.

Table 5: DIPsummaryp.nc component for negotiation of the length of a hash.

```
1       command void* DipSend.getPayloadPtr() {
2       // returns NULL if message is busy
3       if(busy) {
4       return NULL;
5       }
6       return call NetAMSend.getPayload(&am_msg, 0);
7       }
8
9       command error_t DipDecision.send() {
10      dip_msg_t* dmsg;
11      dmsg = (dip_msg_t*) call SummarySend.getPayloadPtr();
12
13      dmsg->type = ID_DIP_SUMMARY;
14      ... code omitted ...
15      }
```

5.3.2 Bug#2: Out-of-bound array error

Another out-of-bound error exists in *DIPSummary.nc* component. Table 6 shows part of a code extracted from *findRangeShadow()* task. This code calculates left and right indices of *ShadowEstimates* array in order to provide a calculation of hash in subsequent lines.

Table 6: Part of DIPSummary that carries Bug#2.

```
1    for (i = LBound ; i + len < RBound ; i++) {
2       est1 = shadowEstimates[i];
3       est2 = shadowEstimates[i + len];
4     }
```

In line 3 of table 6, *est2 = shadowEstimates [i + len]* access is out-of-bound, when an amount of *RBound* value violates a proper one. In this case, it gains a faulty value instead. KLEE found an improper value for *est2*. Table 7 shows more lines of the component code carrying this error. KLEE assigns *ShadowEstimates* a range of values including *UQCOUNT_DIP (128).* We analyzed the log files generated by KLEE. They show that an initial value of high Index is 0, which provides *len = 128* and *LBound = RBound = 128*.

Program continues until value of high index reaches 121. Consequently, *len=8, LBound= 114* and *RBound= 129*. KLEE logs show the value of *RBound* is still 130. Hence, it becomes greater than the maximum current value of 127. This results in out-of-bound error. This error was not detected in memory safe execution phase (with Safe TinyOS tool).

To solve this issue, a part of the code carrying the bug, may have a change as below:

est= shadowEstimates [i+len] to *est2=shadowEstimates [i+len-1]*

Whenever, *RBound* reaches *UQCOUNT_DIP*, no error appears. However, there is a probability of exceeding *UQCOUNT_DIP*. This may provide another safety violation in the code.

Table 7: DipSummaryP code.

```
1     // repeats through the range
2    runEstSum = highEstSum;
3    printf("DipSummaryP:Iterating from %u to %u\n", LBound, RBound);
4     for(i = LBound ; i + len < RBound; i++) {
5       est1 = shadowEstimates[i];
6       est2 = shadowEstimates[i + len];
7       runEstSum = runEstSum - est1 + est2;
8       printf("Dissemination: Next sum: %u\n", runEstSum);
9         if(runEstSum > highEstSum) {
10        highEstSum = runEstSum;
11        highIndex = i + 1;
12        printf("DipSummaryP: Next range: %u, %u = %u\n", highIndex,
13        highIndex + len, highEstSum);
14         }
```

```
15      }
```

5.3.3: Consistency in dissemination

Table 8 shows a coverage provided by KLEE in one execution period. An execution lasts for 5 hours and non-deterministic event that we applied is node failure. As it is shown, during execution approximately 98% of the code is covered by KLEE. In addition, 5% of the code needs to be solved regarding the emitted errors.

Table 8: KLEE coverage statistics in dissemination protocol.

Path	Instrs	Time(s)	ICov (%)	BCov (%)	ICount	Solver (%)
klee-last	3190515	18051.38	98.55	79.96	156242	5.33

Despite of two aforementioned bugs, logs generated by KLEE present a successful eventual consistency of network in case of node reboot and node failure. To test the behavior of DIP protocol toward node reboot, we consider both symmetric and asymmetric network topology. These topologies were described in Chapter 4.

We carefully analyzed an execution path, in which one node is not available. KLEE injects two inputs of different size to network and writes all logs to separate files. Node reboots and rejoins the network. State files in KLEE indicate that a newly joined node has the same shared value after dissemination terminates. When a node reboots symbolically by KLEE, network disseminates different updates for a shared variable, periodically. However, when node rejoins the network, it receives a recent new value and agrees on keeping that. This provides promised consistency in the network.

In the next section, we evaluate TYMO routing protocol. We also show that symbolic execution method (KLEE) detects some bugs while compiler and simulation test tools could not detect them.

5.4 Case 2: TYMO protocol

As discussed in Chapter 2, TYMO is a point-to-point routing protocol for MANET. In the following section, we integrate KLEE with TYMO to test functionality of this protocol. Similar to dissemination, we consider two different network topologies. In one of them, all nodes are connected and network is symmetric. In the second one, we have an asymmetric network to try to study what happens to packet transmission, if a link between two nodes fails. Figure 12 shows all components inside TYMO and an order in which a packet transmits to reach a destination. It is sufficient to go through all those components involved in packet transmission to ensure coverage.

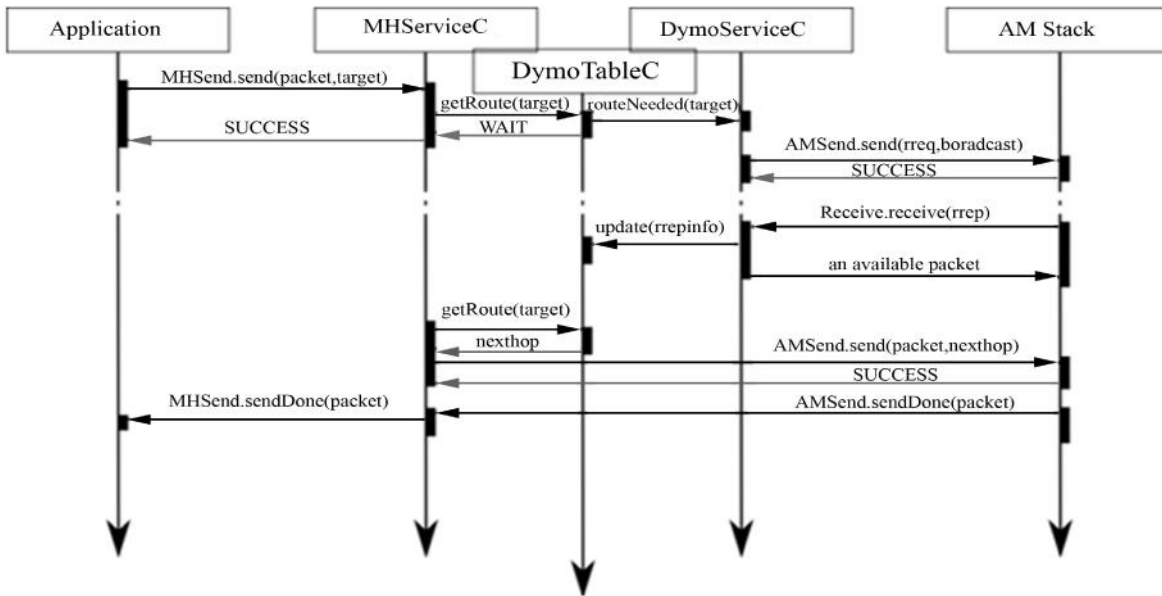

Figure 12: Component of TYMO [30].

Figure 13, demonstrates a network of four nodes as an example. All nodes are connected and communicating well. Execution time of KLEE is set to 5000 seconds, to limit execution paths to time and to control CPU usage and delay. However, at a final evaluation, we let the system to run for 36 hours. In this scenario, we aim to test the robustness of packet transmission against non-deterministic event of packet loss.

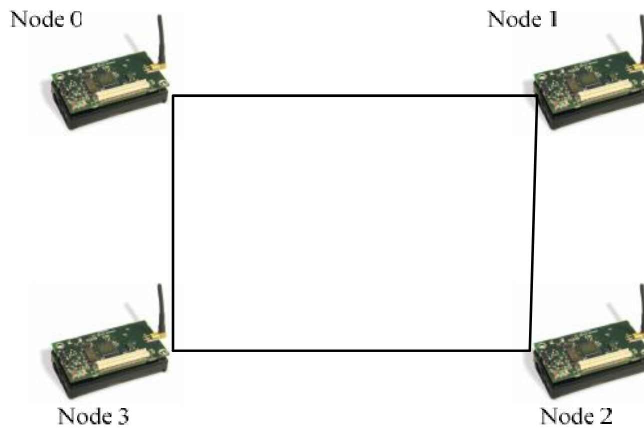

Figure 13: Symmetric network topology

We put distributed assertion for not receiving an acknowledgment. If a sender does not receive an ACK of the sent packet within a specific time declared in the code, it violates the assertion. KLEE then writes error in an output file.

In this execution, we aim to have node numbers and connectivity of nodes as symbolic to test the behavior of the network toward a growing network. We also aim to study node failure effect. KLEE symbolically defines number and ID for nodes in network. The following sections describe how KLEE detected a bug in node ID and ACK handler event.

5.4.1 Bug#1: Node ID 0 failure

This bug is related to behavior of nodes depending on whether they are origin, target or intermediate. KLEE starts from topology depicted in Figure 13 and continues until reaches 20 nodes. Execution time varies between 10 minutes to 36 hours, depending on size of the network. KLEE starts assigning nodes as origin and target. Log files show that when node ID 0 attempts to send a packet to any other nodes, packet transmission fails. However, other nodes with ID of 1 to 100 can successfully finish transmission.

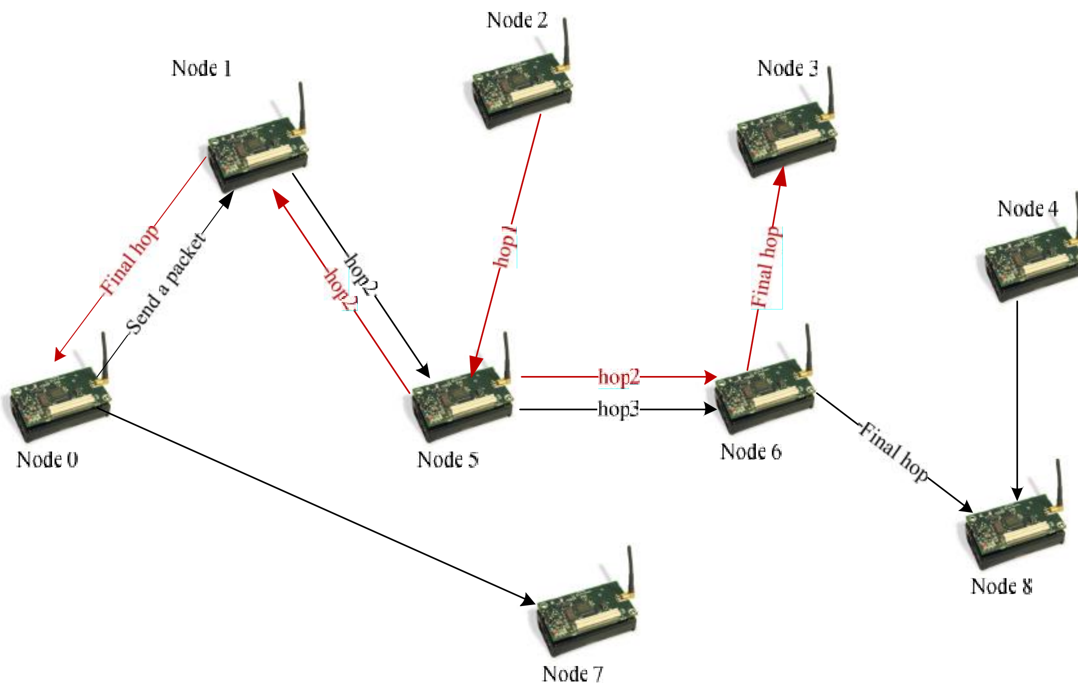

Figure 14: Asymmetric topology of network (implemented by KLEE). Node 0 is origin in black routes and target in red routes.

We found this error by defining a distributed assertion in the component of ACK receipt. If a sender does not receive an ACK of a sent packet, within a specific time, then it violates the assertion. Then KLEE generates an error output file. These files show that all messages in which

origin has ID 0 failed. Those messages are being retransmitted again and again by origin until time limit expires.

In that faulty execution path, no packet is sent and no node received any packet. Therefore, KLEE terminates the path with error. However, execution continues in other paths in which other IDs are origin. For example, Figure 14 shows an asymmetric network composed of 9 nodes. KLEE examined all possible combinations of node interactions.

Suppose set of (origin, destination). Here is an extract from KLEE outputs which shows the path, packet has passed:

(1, 0): [node 1, **node 0**]
(1, 5): [node 1, node 5]
(1, 2): [no route available]
(2, 8): [node 2, node 5, node 6, node 8]
(2, 0): [node 2, node 5, node 1, **node 0**]
(2, 7): [node 2, node 5, node 1, **node 0**, node 7]
(0, 1): [general error occurred]

And so on. As it appears, node with Id 0 can be as target and intermediate node (to transfer a packet to the next hop) without any error.

5.4.2 Bug#2and #3: ACK handling

This bug in TYMO is associated with messages transmitting process. This process is implemented in *ForwardingEngineM* component. We showed the functionality of *ForwardingEngineM* module in Figure 15.

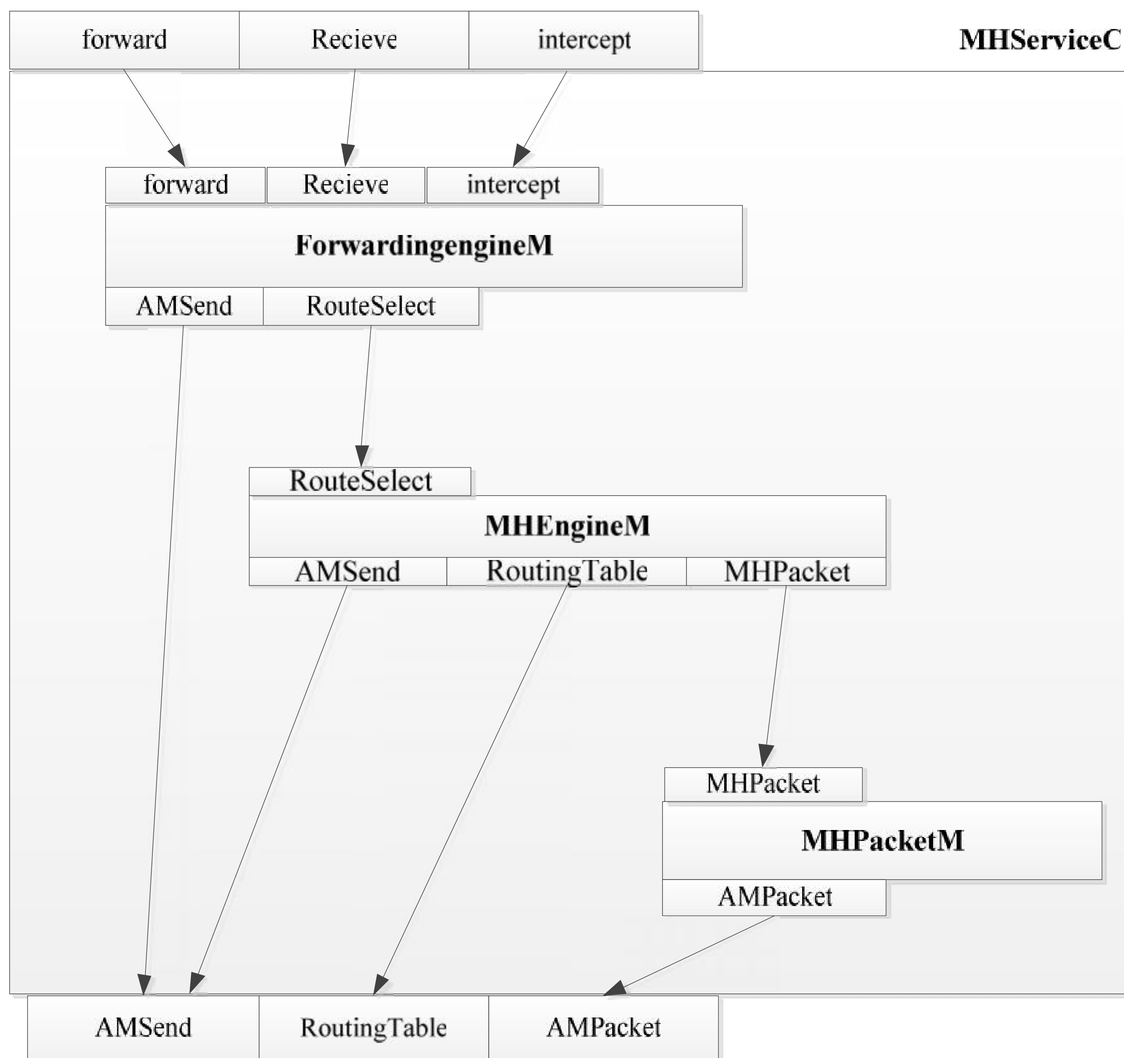

Figure 15: ForwardingEngineM module [13].

ForwardingEngineM module has a simple role to forward a message. As depicted in Figure 15, in a node, a packet is received through *forward* command. Routing engine decides what operations should be done on the packet. Hence it inspects the packet according to protocol and decides to forward that packet by updating it.

Protocol informs the forwarding engine about the decision of the routing engine later. If processing a packet takes a long time, then routing engine makes decision on its own. It decides to immediately drop a packet, process and forward the packet itself or return error message by *AMSend* interface. Bug #2 in TYMO, is related to *ForwardingEngineM* functionality.

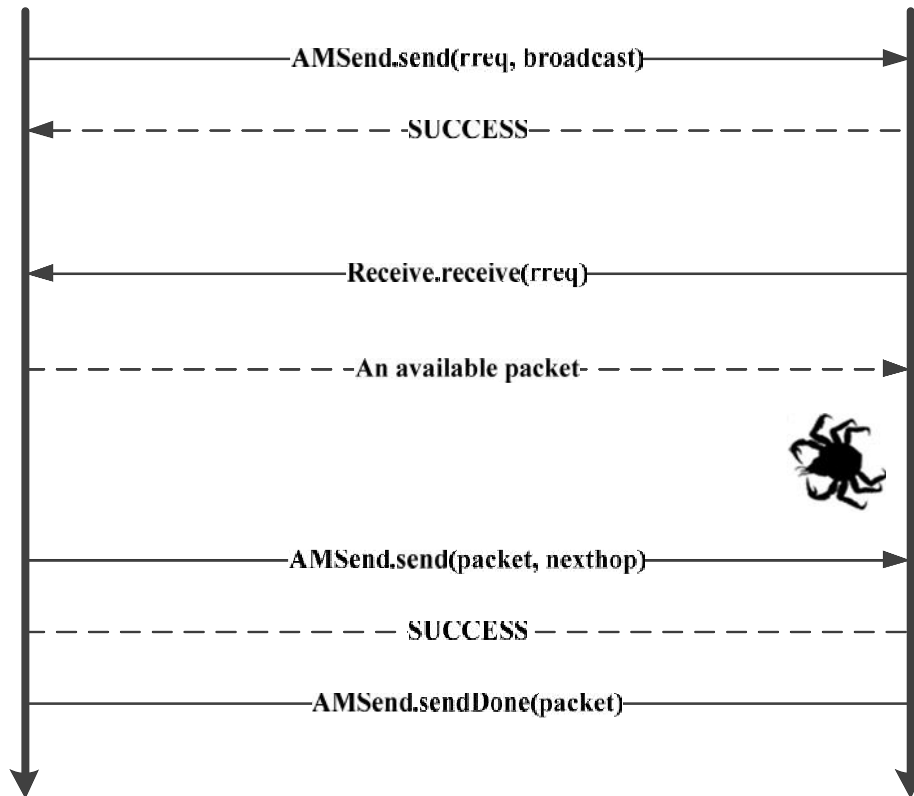

Figure 16: A bug in AMSend.

When the protocol starts, in one execution path, we activate ACK. A node tries to send a message while it does not have a valid route to a destination. As an example, in Figure 14, node 5 joins network and wants to send packet to node 7. There is no direct route between these nodes. The only way is {5, 1, 0, 7} and this entry is only produced in nodes after transferring some messages. A small routing table inside nodes shows that entry of routing table is empty or old when the node joins. This case specially, occurs after KLEE connects a rebooted node to the network. While node was down, routing table of neighboring nodes has been updated with no route to and from the affected node. Hence, we have a successful ACK activated environment. In this case, retransmission timer initiates and *Timer.fired()* starts successfully.

However, if execution lasts for half a day, for example, valid routes are available toward all nodes. Therefore, message is sent without the ACK request activated. There is no way to make sure that whether packet is received by a destination or has been lost in way.

Table 9: SendDone event

```
event void MHSend.sendDone(message_t * msg, error_t e){
  if((e == SUCCESS) && (msg == &packet) && (call MHPacket.address ()
    == ORIGIN) {
    // Even better!
  } else {
    // The packet couldn't be sent!
  }
}
```

Table 9 shows part of a component, *SendDone* event, carrying second ACK handling error. This part of the code is extracted from *MHPacket* package. Figure 16 demonstrates where in communication, the bug exists. Error handling mechanism in *SendDone* indicates that, if for any reason, *e=FAIL* then packet has not been sent. However, log files of KLEE show that even if a packet is sent to the destination, *e=!SUCCESS* occurs. Origin node retries sending packet for duration of time limit. This time is specified in KLEE. Resending the packets causes bandwidth traffic and packet retransmission. Interestingly, we did not detect this bug when we executed the protocol for a short time. This error emerges when we have many nodes running the protocol for a long time. In our case, it lasts 33 hours. We conclude that ACK is not efficiently handled in the protocol in increasing load.

5.4.3 Coverage

Table 10 shows the coverage of TYMO application by KLEE in one execution run. In this execution, KLEE detected the bug discussed in Section 5.4.1. Table shows that KLEE covered 1562541 instructions, equivalent to 28% of total at a very short time of 2.57 seconds. At this short time, KLEE found node ID fault in code.

Table 10: KLEE coverage statistics in one execution in TYMO

Path	Instrs	Time(s)	ICov (%)	BCov (%)	ICount	Solver (%)
klee-last	1562541	2.57	28.02	15.43	15531	21.40

In conclusion, although developers provide a good proof that protocols are working securely enough, there is a need to test and debug the protocol comprehensively. Since, for example, bugs described in this thesis successfully passed compilation and simulation testing phase.

Chapter
6

Conclusion

In this thesis work, we presented a technique of symbolic execution to find bugs in wireless sensor networks. We implemented and prototyped the integration of KLEE with TinyOS, in order to automate testing of sensor networks. As discussed in the first chapter, it is important to completely test embedded applications in a wireless sensor network. We should consider all possible inputs to applications to test the reaction and behavior of the network. All these efforts are important, since if an error remains undetected, troubleshooting and solving phase become costly.

In a wireless sensor network, events such as node outage, packet drops or packet corruption cause complex interaction bugs. Moreover, distributed and resource constraint nature of sensor nodes demand numerous testing in different phases. With KLEE, as a bug hunting tool before deployment, we have an automated bug finding tool. This closes gaps between testing community and developing part. Since KLEE successfully injects non-deterministic events to the network, it finally provides high-coverage to codes and applications. We showed that we could successfully integrate KLEE with TinyOS. KLEE fully covered codes and found some bugs.

We used KLEE to test two sample protocols implemented in TinyOS in distributed scenarios. The scenarios that we considered in evaluation and implementation phase, consist of 3 to 100 nodes. KLEE decides how many nodes are there in network at any time. These nodes are connected to each other at start and at a certain time, one or more nodes are disconnected. This helped us to study an effect of node outage event in network.

KLEE found some memory faults and out-of-bound pointer/array in DIP and TYMO. To deeply check applications, we used assertion to verify the condition of a code, in case assertion violates. Then we replayed and analyzed the failure scenario to study what caused assertion to violate. This led us to find some more bugs. By integrating KLEE into two protocols, dissemination and TYMO, we found an ACK handling problem, a node ID and hash faults.

6.1 Limitations

We used KLEE execution model in this thesis work. It requires a testing system with powerful processors and memory storage to handle a large number of log files and error logs. Hence, a comprehensive evaluation requires a real super computer. Moreover, more protocols should be

evaluated with different topologies to make a better judgment on performance of KLEE in TinyOS. A real environment consists of hundreds of nodes. Therefore, a more comprehensive debugging requires providing such an environment. In this thesis work, we provided a dimension of 20 nodes in DIP and 100 nodes in TYMO. However, in real implementations, network is much larger.

One other limitation is that, during test execution, run time and memory usage depends on symbolic inputs and a number of inputs. These factors are defined by KLEE. As discussed earlier, defining non-deterministic events as symbolic affects the number of execution paths. Therefore, even with a few number of inputs or a small packet injected to the network, we face numerous execution paths. This limitation was greatly solved by distributed assertions, since they reduce a number of paths. One important necessity to use KLEE is that we need to have knowledge of the application logic before applying KLEE on it.

6.2 Future work

It is important to integrate KLEE with more protocols such as CTP, which is broadly used. In addition, we only considered DIP; hence other dissemination libraries need to be considered later. Another future work is to incorporate more checks in KLEE such as runtime monitoring of intensive computational tasks and monitoring of long loops. Another demand is to apply time annotations. Moreover, interested people can integrate KLEE with other operating systems and development platforms of wireless sensor networks. They can also address and verify another distributed behavior of sensor protocols such as state transition.

References

[1] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli. The Hitchhiker's Guide to Successful Wireless Sensor Network Deployments. In SenSys, 2008.

[2] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and Yield in a Volcano Monitoring Sensor Network. In OSDI, 2006.

[3] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A Macroscope in the Redwoods. In SenSys, 2005.

[4] K. Langendoen, A. Baggio, and O. Visser. Murphy Loves Potatoes: Experiences from a Pilot Sensor Network Deployment in Precision Agriculture. In WPDRTS, 2006.

[5] R. Sasnauskas, O. Landsiedel, M. HamadAlizai, C. Weise, S. Kowalewski, and K. Wehrle. KLEENet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In Proc. of the 9th ACM/IEEE International Conference on Information Processing inSensor Networks (IPSN), Stockholm, Sweden, April 2010.

[6] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In OSDI, 2008.

[7] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS), 2006.

[8] http://klee.llvm.org/.

[9] http://www.tinyos.net/tinyos-2.x/doc/html/tep118.html.

[10] http://docs.TinyOS.net/tinywiki/index.php/Dissemination.

[11] K. Lin and P. Levis. Data Discovery and Dissemination with DIP. In IPSN, 2008.

[12] 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. In Ambient Intelligence, 115-148. 2005.

[13] http://docs.tinyos.net/tinywiki/index.php/Tymo.

[14] N. Kothari, T. Millstein, and R. Govindan. Deriving State Machines from TinyOS Programs Using Symbolic Execution. In IPSN/SPOTS, 2008.

[15] S. Gupta, R. Zheng and A. M. K. Cheng, "ANDES: an Anomaly Detection System for Wireless Sensor Networks," Proceedings of the IEEE International Conference on Mobile Adhoc and Sensor Systems, 1-9, 2007.

[16] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In SenSys, 2003.

[17] L. Luo, T. He, G. Zhou, L. Gu, T. F. Abdelzaher, and J. A. Stankovic. Achieving Repeatability of Asynchronous Events in Wireless Sensor Networks with EnviroLog. In INFOCOM, 2006.

[18] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level Sensor Network Simulation with COOJA. EWSN, 2006.

[19] J. Polley, D. Blazakis, J. McGee, D. Rusk, J. Baras, and M. Karir. ATEMU: A Fine-grained Sensor.

[20] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo. Declarative Trace points: A Programmable and Application Independent Debugging System for Wireless Sensor Networks. In SenSys, 2008.

[21] V. Krunic, E. Trumpler, and R. Han. NodeMD: Diagnosing Node-level Faults in Remote Wireless Sensor Systems. In MobiSys, 2007.

[22] K. Romer and J. Ma. PDA: Passive Distributed Assertions for Sensor Networks. In IPSN/SPOTS, 2009.

[23] T. Sookoor, T. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macro debugging: Global Views of Distributed Program Execution. In SenSys, 2009.

[24] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr. Surviving Sensor Network Software Faults. In SOSP, 2009.

[25] P. Ballarini and A. Miller. Model Checking Medium Access Control for Sensor Networks. In ISOLA, 2006.

[26] P. C. Olveczky and S. Thorvaldsen. Formal Modeling, Performance Estimation, and Model Checking of Wireless Sensor Network Algorithms in Real-Time Maude. Theor. Comput. Sci., 410(2-3):254-280, 2009.

[27] P. Volgyesi, M. Maroti, S. Dora, E. Osses, and A. Lfiedeczi. Software Composition and Verification for Sensor Networks. Sci. Comput. Program., 56(1-2):191-210, 2005.

[28] P. D. K. Varma and V. Radha, “Prevention of Buffer Overflow Attacks Using Advanced Stackguard,” Proceedings of the International Conference on Advances in Communication, Network, and Computing (CNC), 2010.

[29] M. M. H. Khan, L. Luo, C. Huang and T. F. Abdelzaher, “Snts: Sensor Network Troubleshooting Suite,” Proceedings of the 3rd IEEE International Conference DCOSS, Springer LNCS, 45-49:142-157, 2007.

[30] D. Gay et al. The nesC language: a holistic approach to networked embedded systems. In Programming Language Design and Implementation (PLDI’03), 2003.