Sequence Mining Based Debugging of Wireless Sensor Networks

A Thesis Presented for

The Master of Science

Degree

The University of Tennessee, Knoxville

Kefa Lu

May 2013

Acknowledgements

I would like to extend my sincere gratitude and appreciation to all the individuals who made this thesis possible.

First and foremost, I would like to thank my advisor, Dr. Qing Cao for assisting and encouraging me and for always his numerous suggestions in my research. Without his excellent guidance and support, this work would not have been possible. I also want to give sincere thanks to Dr. Thomason and Dr. Gao. I greatly appreciate their precious time and significant input to this thesis. I would like to give my special thanks to my parents, my parents-in-law, my wife and my two kids, who provide the encouragement and the strongest support for my study in every possible way.

Finally, I thank all the members in the Lanterns Lab for their useful advise and suggestions presented at meetings and discussions.

Abstract

Wireless Sensor Network (WSN) applications are prone to bugs and failures due to their typical characteristics, such as extensively distributed, heavily concurrent and resources restricted. It becomes critical to develop efficient debugging systems for WSN applications. A flexible and generic debugger for WSN applications is highly demanded. In this thesis, I proposed and developed a flexible and iterative WSN debugging system based on sequence analyzing and data mining techniques. At first, I developed vectorized Probabilistic Suffix Tree (vPST), a variable memory length model to extract and store sequential information from program runtime traces in compact suffix tree based vectors, based on original Probabilistic Suffix Tree (PST). Then I built a novel WSN debugging system by integrating vPST with Support Vector Machine (SVM), a robust and generic classifier for both linear and nonlinear data classification. The vPST-SVM debugging system enables developers to target at any hot spots they know might be problematic in the program source codes. They simply need insert trace points into the hot spots, collect runtime traces, then iteratively analyze the traces and finally locate real bugs. At last, I studied three different test cases, two on LiteOS and one on TinyOS, to evaluate the proposed WSN debugging system. It is demonstrated to be an efficient, flexible, generic and portable WSN debugger by the case studies. In addition, the vPST-SVM sequence analyzing methodology provides researchers with an inspiring angle of view on extracting sequential features and obtaining meaningful insights from sequences by appropriate transformation of sequential data.

Contents

1 Introduction **1**

2 Debugging Wireless Sensor Networks **4**

2.1 WSN Debugging Techniques 4

2.2 Logging and Symptom Mining 5

3 WSN Debugging by Sequence Mining **8**

3.1 Sequence Mining and WSN Debugging 8

3.2 Theoretical Background 9

3.2.1 Probabilistic Suffix Tree 10

3.2.2 Support Vector Machine 12

4 Vectorized Probabilistic Suffix Tree **14**

4.1 Limitations of Traditional Probabilistic Suffix Tree 14

4.2 Vectorized Probabilistic Suffix Tree 15

4.3 vPST-SVM Iterative Debugging System 16

5 vPST-SVM Debugger Design and Implementation **19**

5.1 System Design 19

5.2 System Implementation 23

5.2.1 Trace Collector 23

5.2.2 Preprocessor 23

5.2.3 vPST Analyzer 24

5.2.4 SVM Classifier 24
5.2.5 Developers 25

6 Evaluations **27**

6.1 Test Case I: a variable overflow bug in a LiteOS application 27
6.2 Test Case II: a race condition bug in a LiteOS application 31
6.3 Test Case III: a race condition bug in a TinyOS application by iterative debugging 32

7 Conclusions **36**

Bibliography **37**

Vita **43**

List of Tables

6.1 Bug 1 trace analyzing results 28
6.2 Bug 2 trace analyzing results 32
6.3 Bug 3 trace analyzing results 33

List of Figures

3.1 Probabilistic Suffix Tree 11

4.1 a Vectorized Probabilistic Suffix Tree 16

5.1 System Architecture 21

Chapter 1

Introduction

In the past decade Wireless Sensor Networks (WSNs) have been widely studied, developed and deployed for various purposes, including surveillance, environmental monitoring and data collection Kulakov and Davcev (2005), Kwon et al. (2004), Langendoen et al. (2006). Intensive research has been conducted on WSN design and development. However, WSN applications are still suffering from hidden bugs and frequent failures Langendoen et al. (2006), Werner-Allen et al. (2006), due to their typical characteristics, such as distributed architecture, concurrent execution model and strict resource limitations. It is very difficult to perform efficient debugging on WSN applications, because most of them are context sensitive and event driven. It is usually infeasible to fully control their operating context and triggering events. As a result, many WSN bugs are transient and irreproducible. Ramanathan et al. (2005) It becomes a big challenge for current WSN researchers and developers to design and develop a robust WSN debugging system that is able to efficiently detect hidden and transient bugs.

The motivation of this thesis is to develop a robust and generic WSN debugging system. Regarding diagnosis of WSN with transient and context sensitive bugs, program runtime traces from real deployment are the most valuable data compared to source codes and simulation traces, because they contain the most precious

information of program runtime states and associated contexts in actual operations. In fact, many transient WSN bugs can only be triggered by specific contexts. Ramanathan et al. (2005) They can only be caught in program runtime traces and logs, but can hardly be found in their source codes or simulations. In fact, most bugs change program control flow during runtime. In most cases it is just the changing of program control flow that leads programs to wrong states and yield unexpected results. Jiang and Su (2007) So it is crucial to build a WSN debugging system that can analyze runtime traces efficiently.

The most critical challenge in designing and building such a system is how to extract and analyze sequential information from program runtime traces efficiently, since program runtime traces are naturally sequences that can be collected during runtime. To reveal hidden bugs, the debugging system needs to be able to obtain meaningful insights from runtime traces. So it is necessary to build models that can efficiently extract sequential information from sequences and represent it in compact form that can be easily analyzed.

In this thesis, I designed, implemented and evaluated a flexible and generic debugging system Lu et al. (2012) based on sequential data analyzing and outlier detecting techniques, including two theoretical models, vectorized Probabilistic Suffix Tree (vPST) and Support Vector Machine (SVM). Original PST model is a flexible probabilistic model that can efficiently extract and store sequential information from sequences in compact suffix tree data structure,Ron et al. (1996) while SVM is a robust and generic classification technique that can solve both linear and nonlinear classification problems.Aizerman et al. (1964) By extending PST to vPST, we are able to not only retain sequential information but also significant substructures within sequences in compact and simple vectors. SVM can be easily applied on the vectors to detect outliers in the sequences. By combining vPST model and SVM classifier together with an efficient tracing subsystem, I designed a flexible and generic debugging system for WSN applications by iterative sequence mining.

The contribution in this thesis is three-fold. First, I extended PST model to vPST model and integrated it with SVM to make an robust outlier detecting system that can not only efficiently extract sequential information from sequences, but also can perform decent classification on them. The proposed vPST-SVM sequence outlier mining system was demonstrated to be a novel and efficient anomalous sequence detecting system. Second, the study on the system and its performance by different test cases provided researchers on sequence analysis with a new angle of view on methodology development of extracting and analyzing sequential information. The vPST model gives researchers inspiration on exploring new sequence analyzing approaches by breaking sequence into pieces and storing them in some meaningful data structures for efficient analyzing. Third, the proposed WSN debugging system was demonstrated to be a new and effective debugging system suitable for WSN applications, especially for detecting transient bugs. The iterative debugging approach was evaluated by showing prediction results from different test cases developed on different operating systems with incremental changing of vPST depth and iterative debugging cycles. These results shed light on WSN debugging systems design and development for future researchers.

The following of this thesis is organized as follows. In chapter 2, I briefly discuss wireless sensor networks debugging techniques, including proposed models and systems. In chapter 3, I discuss background on sequence mining based debugging techniques, including theoretical background on PST model and SVM classification approach. In chapter 4, I describe details on the vPST model and the iterative vPST-SVM anomaly detecting approach. In chapter 5, I describe the system design and implementation. In chapter 6, I present three interesting test cases for system evaluation. I demonstrate the robustness, flexibility and portability of our proposed debugging system by the case studies. Chapter 7 is conclusions of this thesis with some discussions.

Chapter 2

Debugging Wireless Sensor Networks

2.1 WSN Debugging Techniques

There are numerous debugging techniques proposed and developed over the past decade. Li and Regehr (2010); Sasnauskas et al. (2010); Zhou et al. (2010) The various WSN debugging techniques can be classified based on their usage in application life cycle as pre-deployment, deployment-time and post-deployment validation, or based on their implementation strategies as software-based, hardware-based and hybrid. Sreedevi and Sebastian (2012); Schoofs et al. (2012).

Pre-deployment debugging tools are used for debugging WSNs prior to actual deployment. There are software debuggers, software simulators, software emulators, and testbeds. Deployment-time tools validate system functionality at the time of deployment to lower the risk of early failures. SeeDTV Liu et al. (2007) is an example of deployment-time tool that minimizes requirement of revisiting deployment contexts which are composed of environmental states that are difficult to replicate. Post-deployment debugging tools are used for debugging WSNs post actual deployment. Such tools collects and analyze information from program runtime such as packets

sent and received between different nodes, program execution logging and tracing data and etc.

Different debugging tools use different implementation strategies in practical applications. Some debugging tools are software-based, such as software simulators and software emulators. Some debugging techniques are implemented on real hardware, including some record and reply debugging techniques. There are also many debugging techniques implemented as a hybrid of both software and hardware.

2.2 Logging and Symptom Mining

In this thesis, I mainly focus on analyzing program runtime traces from post-deployment WSN applications by software-based implementations. First of all, program runtime logs and traces contain the most valuable program execution and contextual details. Such runtime details are the most important sources for detecting bugs and root causes of those bugs. In WSN applications, there are many transient bugs that can hardly reproduced or replicated due to rare occurence of their triggering contexts. So recorded program runtime traces become the most reliable sources for studying these transient bugs once triggered. Secondly, program runtime traces generated from real deployment under real environmental conditions and contexts may be difficult to be replicated from pre-deployment simulations of emulations, because people may not have enough prior knowledge of triggering conditions and contexts for replicating some bugs under study. Compared with pre-deployment simulations and emulations, post-deployment runtime traces provide much more objective information on program execution and behaviour.

Even there are many different debugging systems designed and developed, there is still a significant lack of efficient program runtime trace debugging system that can fully take advantage of runtime traces from real deployment. Some of them were designed with very low portability due to their limitation to specific operating systems Zhou et al. (2010); Li and Regehr (2010). Some others of them were restricted to

source code analysis or simulation trace analysis Li and Regehr (2010); Sasnauskas et al. (2010). There are many tricky bugs caused by race conditions or inappropriate controlled concurrencies that can only be triggered in real deployment under some specific circumstances. Also, there are many different types of WSN applications developed based on many different well developed WSN operating systems, such as TinyOS, LiteOS, Contiki and etc. Thus, a flexible, portable and generic WSN debugging system that can work efficiently with different WSN operating systems is highly demanded. Following is a brief discussion of two representative debugging systems based on logging and symptom mining to reveal the most significant limitations of recently proposed trace analyzing based WSN debugging systems.

Dustminer Khan et al. (2008) is a debugging system based on frequent patterns mining. Its data collecting front-end performs runtime logging by collecting events from distributed sensors within a network. Then it takes predefined "good" patterns and "bad" patterns by application developers to separate the collected data into two piles, a "good" pile and a "bad" pile. Consequently, its data analysis back-end applies frequent patterns mining approaches to detect frequent subsequences in both piles. Finally it tries to identify the probable causes of failures by comparing frequent patterns between "good" piles and "bad" piles. There are three significant drawbacks of this approach. First of all, it is based on and limited to frequent patterns mining. So for those failures or bugs that could only be triggered by some rare events, obviously Dustminer will fail to detect them since this kind of bugs can only generate infrequent patterns. Secondly, it requires a lot of human effort from application developers to figure out clear definitions of "good" patterns and "bad" patterns for data training. As a matter of fact, before the bug is found, it is usually not practical to define clearly what are "good" patterns and what are "bad" patterns. So the large amount of overhead caused on developers practically makes it difficult to apply this approach in actual application development. Third, frequent patterns mining is usually very expensive if the sequence is long.

Sentomist Zhou et al. (2010) is another interesting debugging system designed for WSN applications. It was specifically designed for TinyOS applications. It collects TinyOS application runtime traces from simulations by a complicated procedure. Initially, it partitions collected traces into smaller event handling intervals, which is only valid on TinyOS. Then it analyzes instruction counters of all of such event handling intervals. By one-class SVM classifier, it sorts all collected event handling intervals based on the probabilities they are abnormal or normal. This debugging system suffers from several major drawbacks. At first, it is strongly restricted to TinyOS operating system. Its event handling interval model is specifically based on event-driven execution mechanism of TinyOS. Secondly, it can only collect application execution data from simulations. In fact, there are many bugs that can only be triggered in actual deployment by specific environmental conditions, but not in simulations. Thirdly, it uses only instruction counters, so all sequential information is totally ignored. For example, two different sequences (1, 1, 2) and (2, 1, 1) can be considered identical if they are converted to instruction counters, both of which are (1:2, 2:1). In addition, even Sentomist finally marks some event handling intervals that may contain bugs with very high probabilities, it is still too hard for a developer to manually check a sequence with possibly several hundreds of instructions. It is unrealistic for human being to manually check those long sequences.

Chapter 3

WSN Debugging by Sequence Mining

3.1 Sequence Mining and WSN Debugging

As discussed in previous chapter, this thesis is focused on analyzing WSN application runtime traces from post-deployment to detect hidden bugs and failures. Those runtime traces are naturally sequences of execution of program source codes. Therefore, the problem of debugging WSN by analyzing runtime traces reduces to a more abstract problem of detecting anomalies from sequences, i.e. a typical sequence mining problem. How efficiently a sequence mining system can detect anomalies in collected runtime traces directly determines how efficiently this system can be used as a WSN debugger.

In principle, an efficient and effective sequence mining based debugging system need be able to play two functional roles very well. At first, it should be able to analyze and extract sequential information within sequences efficiently. Secondly, it should be able to differentiate anomalous sequences from normal ones reasonably well to be qualified as an effective WSN debugger. The two currently proposed WSN debuggers, Dustminer and Sentomist reviewed in previous chapter are the two most interesting

sequence mining based WSN debuggers proposed in past years. Dustminer debugs WSNs by mining frequent patterns within program runtime logs, while Sentomist debugs WSNs by detecting anomalous subsequences from program simulation traces by transforming subsequences into instruction counters. Both systems work for some cases. However, neither of them optimally take advantage of sequential information within program traces or logs. Frequent patterns mining is very expensive, and instruction counters ignore sequential information within subsequences. In this thesis, an innovative sequence mining and analyzing approach is proposed to build a more robust WSN debugging system.

3.2 Theoretical Background

Classical PST model Ron et al. (1996) and SVM classification model Vapnik (1995) are two fundamental models of the proposed debugging system. PST is an adaptable statistical model that generalizes Markov Chain model. It takes as many events as needed to build a suffix tree based on conditional probabilities of finding some events after a given event sequence. This probabilistic model is more realistic and general than traditional Markov Chain model, which ideally assume any event has only effects on its adjacent neighbor. By building probabilistic suffix trees from collected sequences, much sequential information is conserved and can significantly improve following analysis. SVM is a well-developed and remarkably robust classification technique. It is very powerful with respect to sparse, noisy and high-dimensional data. Most importantly, it is a generic classifier suitable for both linear and non-linear classifications. It is widely used for various classification cases from text categorization in natural language processing to protein function prediction in biological science. Its robustness and generic property make it an ideal choice for our debugging system design.

3.2.1 Probabilistic Suffix Tree

PST model was first proposed in 1996 Ron et al. (1996). In the past 15 years, it has been widely used for modeling various complex sequences, including biological sequences and temporal sequences Leonardi (2006); Liao and Noble (2003); Christine (2003); Mazeroff et al. (2008). As discussed in Ron's seminal paper Ron et al. (1996), PST is fundamentally a probabilistic model with variable memory length for modeling sequences. It is based on a well-accepted observation of sequential data that, given proceeding subsequence with a length L larger than some fixed value, the probability distribution of the next symbol does not change significantly. This is also known as *short memory* property of sequential data. The length L is usually called *memory length*. In practice, the memory length is variable for different sequences. As a result, PST model can adaptively extract sequential information from sequences based on their memory length.

Given a sequence of symbols from an alphabet, a PST can be built based on probabilities of each symbol conditioned on different proceeding subsequences. The data structure will be a classical suffix tree. Each node of the tree contains two types of information, a sequence of symbols and a vector composed of conditional probabilities of each possible symbol in the alphabet proceeded by the sequence. The sequence of a node is generated from its parent node by taking its parent's sequence as it suffix. That is the reason why it is called *Probabilistic Suffix Tree*. The root of the PST is built by calculating unconditional probabilities of each symbol of the alphabet. Nodes on subsequent levels are built by growing sequences from their parent nodes and calculating probabilities of each symbol proceeded by corresponding sequences of the nodes.

Figure 3.1 shows a PST to depth 2 that is constructed from a given sequence (1 3 2 2 3 2 1). All shadowed nodes are with subsequences that appear at least once in the given sequence, while white nodes are empty with no occurrence of the subsequences on them. There are 3 distinct symbols in total, 1, 2, 3. The last element 1 in the

sequence is excluded in PST construction because it cannot be a prefix of any other subsequences, but always be a suffix of some other subsequences. On root node, probability vector is calculated based on unconditional probabilities of each symbol in the sequence. There are 1 occurrence of symbol 1, 3 occurrences of symbol 2 and 2 occurrences of symbol 3 out of total 6 instances. So the resulting probability vector is (1/6 3/6 2/6). After construction of root, first order of PST nodes can be constructed. For example, on node 2, symbol 2 followed by symbol 1 once, followed by symbol 2 once and followed by symbol 3 once, the resulting probability vector thus is (1/3 1/3 1/3). PST of this sequence can grow deeper until the longest subsequence, which is the whole original sequence itself is reached. More detailed description on building a PST could be found in references Ron et al. (1996) and Mazeroff et al. (2008).

There are several important advantages of PST. First, PST is actually a generalization of Markov Chain model by extending chain length to some predefined value L. So it is basically a generic probabilistic model that can be applied on

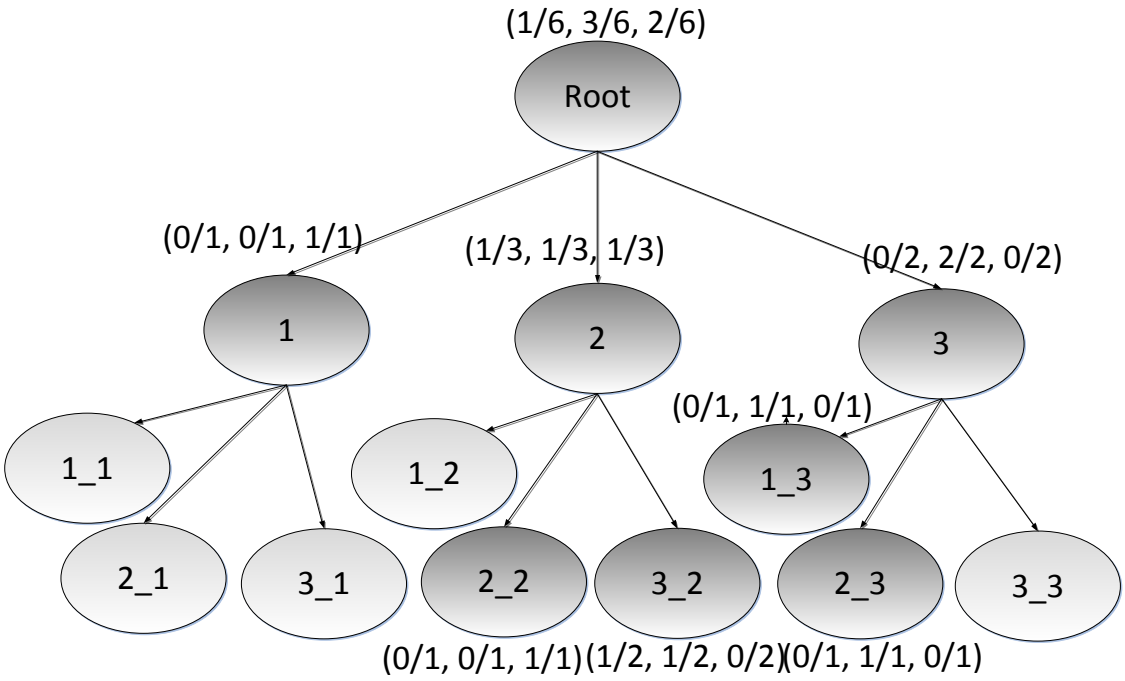

Figure 3.1: Probabilistic Suffix Tree

sequential data. For its utilization on WSN applications, developers could generate any sequences he/she thinks might be relevant, such as function call sequences, instruction sequences, trace point sequences, and etc. Our system will be able to analyze all of such sequential data based on PST model. Second, PST can be used to retain sequential information as much as needed to achieve expected prediction accuracy by adapting it to the specific application in study through memory length adjustment.

3.2.2 Support Vector Machine

The basic idea of SVM is to construct an optimal hyperplane to separate different patterns. By taking advantage of a technique called kernel trick, which was first published by Aizerman et al., Aizerman et al. (1964) SVM can efficiently classify both linearly separable and nonlinearly separable data. It transforms original data from low-dimensional space to high-dimensional space so that a hyperplane could be constructed, and the data can be easily separated. A main advantage of SVM is its ability of analyzing high-dimensional data, which perfectly matches the need of analyzing PST vectors derived from program runtime traces in following WSN debugging system.

One-class SVM Schlkopf et al. (2001) is a variant of SVM that is suitable for outlier detection in a data set with majority of normal data points. Initially it assumes all data points belong to one class, i.e. normal class. Then it finds optimal hyperplane that can represent majority characteristic of the given data set. Based on the optimal hyperplane, majority of the data points that are within one side of the hyperplane are classified as normal, while the remaining data points that are on the other side are outliers. In principle, the relative position between a data point and the hyperplane is a reasonable indicator of the possibility that it is an outlier. The further it is from the hyperplane and on the normal side, it is more likely it is not an outlier. Meanwhile, the further it is from the hyperplane and on the abnormal side, it is more likely it is

an outlier. So the relative position of a point with respect to the hyperplane can be regarded as an anomaly score that indicates its likelihood of being an outlier. This is how the anomaly scores of all WSN program runtime traces are quantitatively calculated from SVM analysis in the WSN debugging system proposed in this thesis.

Chapter 4

Vectorized Probabilistic Suffix Tree

4.1 Limitations of Traditional Probabilistic Suffix Tree

Traditional PST model is an excellent probabilistic model that can represent sequential information within sequences in well-structured suffix trees. However, analyzing PSTs is always challenging, because PSTs constructed from different sequences usually have different structures. For example, how basic analysis like similarity estimation of PSTs with different structures can be determined. In the past decade, different approaches were proposed to analyze similarities between different PSTs Sun et al. (2006). Given two sequences S1 and S2, most proposed approaches, such as the *Odds* and the *Normalized* measure Sun et al. (2006), calculate their similarity based on pure statistic analysis, which is estimation of probabilities of deriving S1 from S2 by estimating probabilities of deriving subsequences of S1 from S2. In other words, these statistics-based analyzing approaches mostly rely on statistic estimation of occurring probabilities of one sequence given occurrence of another one based on occurrence of shared subsequences. Basically, absolute structural similarities are not taken into primary consideration. In fact, probability of deriving one sequence from another is not necessarily equivalent to their absolute structural similarity

but correlation of their conditional possibilities of occurrence of one another. It is plausible to use such probability-based similarities in comparing sequences in some situations, such as program runtime trace analysis for detecting bugs, because non-shared subsequences of difference sequences are ignored.

4.2 Vectorized Probabilistic Suffix Tree

My proposed Vectorized Probability Suffix Tree(vPST) solved above mentioned issue from an new angle of view. It evaluates similarities of different sequences by taking every elements and all possible subsequences up to predefined length into consideration, not just shared subsequences. As is known right now, there is a probability vector on each PST node, which is composed of conditional probabilities of every symbol occurs after the given symbol subsequence of that node. For those subsequences with no occurrence, we can still construct corresponding nodes by putting zeroized vectors on them. In this way, we can construct full PSTs to some predefined depth. Then all these PSTs can be vectorized by combining all separate vectors on all nodes. Take PST in Figure 3.1 for example. It is a full PST with all nodes up to length 2. A vector can be constructed from this PST by sequentially combining probability vectors on all nodes in the order (root, 1, 2, 3, 1_1, 2_1, 3_1, ..., 3_3) as shown in Figure 4.1. In practice, the vectors constructed from such full PSTs even with just a small depth can be very sparse because many nodes may have zero occurrence. Fortunately, the sparse vectors can be compressed by removing zeroized nodes that occur in all PSTs. As a result, a set of standard and uniform vectors can always be constructed from some given set of sequences following the procedure.

Analyzing vPST is much easier than analyzing PST. First, analyzing vectors is much easier than analyzing suffix trees. Absolute structure based similarity estimation of two vectors is more straightforward than that of two suffix trees by simple calculation of Euclidean distances between vectors. Second, the most interesting sequential information and substructures within the sequences can be

retained very well by vPST vectors as by PST, but in simpler data structure. SVM can be applied directly on vPST vectors for outliers detection.

4.3 vPST-SVM Iterative Debugging System

A natural advancement of vPST is to integrate it with a robust classification approach SVM, which is powerful on analyzing high-dimensional data. As a result, we developed vPST-SVM sequence analyzing approach and demonstrated its efficiency by applications in our WSN debugging system. Detailed steps of the iterative vPST-SVM debugging process are listed below.

1. Construct PSTs from given sequences.
2. Vectorize PSTs to generate vPST vectors.
3. Apply SVM on vPST vectors to obtain a short list of top anomalous sequences (usually 10 to 20 sequences) based on calculated anomaly scores.

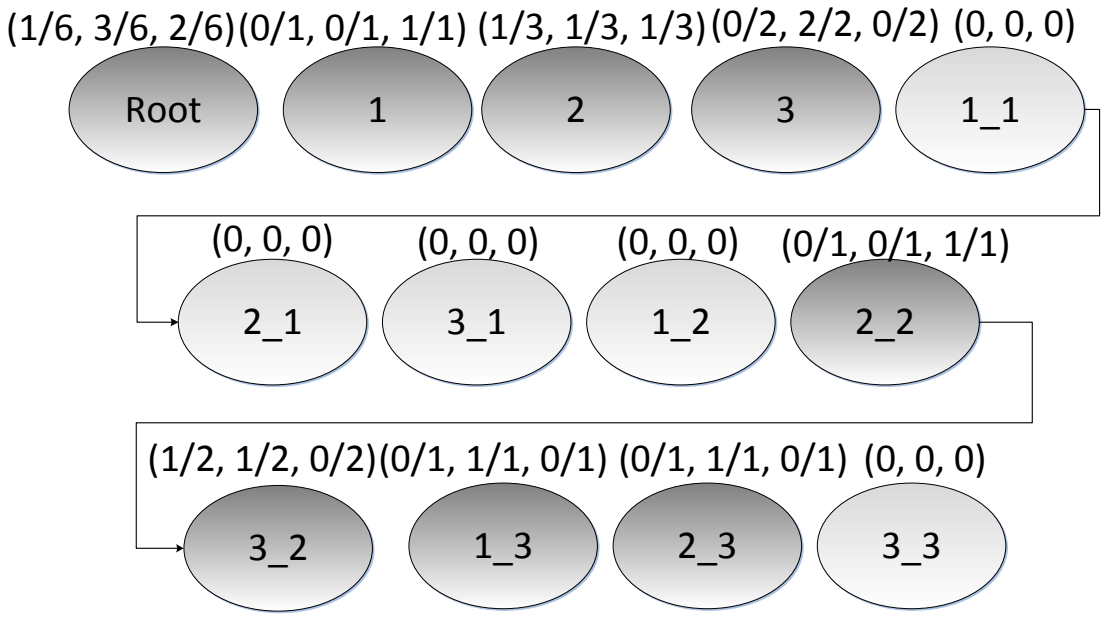

Figure 4.1: a Vectorized Probabilistic Suffix Tree

4. Developer checks the short list of anomalous sequences to confirm them as anomalies or bugs.

5. Learn from identified anomalies by removing confirmed anomalies and go back to step 3.

6. Stop until enough bugs are detected and confirmed.

In the debugging process, vPST depth is a significant parameter that can be tuned adaptively for efficient bugs detection. It can be increased for tricky bugs that are hard to be detected by low-depth vPSTs, meanwhile it can be decreased for simple bugs that are easily detectable to save resources and improve efficiency. In principle, the larger the vPST depth is, the more sequential information is extracted and retained from sequences, and bugs are more likely to be covered by collected patterns in vPSTs. However, increasing of vPST depth also leads to more expensive computing in following analyzing. As is known to all of us, vPST size increases exponentially as its depth increases. So too large of vPST depth is not practical in actual sequence analysis. Fortunately, sequences have *short memory* property, which can guarantee vPSTs with reasonable low depth can still cover majority of sequential information with sequences. In our case studies, reasonable debugging results were obtained even we only explored vPSTs with depth no larger than 5. It indicates for our debugging purpose, a memory length of 5 is enough for retaining majority sequential information from runtime traces.

The vPST-SVM debugging system can iteratively learn from identified anomalies and confirmed bugs by taking feedback from developers. This learning process is simple but very efficient on debugging. It will be demonstrated by test cases. The vPST-SVM system is basically an anomaly detecting system. What it is good at is finding outliers from all input sequences. But not all outliers are necessarily caused by software bugs. With a little feedback from developers by telling the system which top outliers detected are just anomalies, the system can be quickly improved for better bugs detection in following iterations.

The computational complexity of the iterative vPST-SVM debugging process is mainly determined by construction of PSTs and application of SVM on the vPST vectors. The former is approximately $O(Ln \log n)$ based on ref Mazeroff et al. (2008), where L is total number of sequences in study and n is average length of all sequences. The latter is approximately $O(Ln_{SV} + {n_{SV}}^3)$ based on ref Keerthi et al. (2006), where n_{SV} is total number of support vectors. So the total computational complexity of iterative vPST-SVM debugging approach is approximately $O(N(Ln \log n + Ln_{SV} + {n_{SV}}^3))$, where N is number of iterations.

Chapter 5

vPST-SVM Debugger Design and Implementation

5.1 System Design

The system design follows three principles. First, the system needs to be simple and easy to use. The basic motivation of this project is to design and build a powerful debugging tool to ease developmental process by assisting developers in actual testing and debugging processes. So the debugging system should be able to save the effort of developers in practical development. Too much overhead, such as high learning curve or complicated debugging procedure is not preferred. Second, the debugging system needs to be portable. As discussed earlier, a significant issue with many proposed systems is their low portability. Most of the proposed debugging systems were designed and restricted to some specific applications or some specific WSN operating systems. They require significant amount of effort for developers to adapt the proposed debugging systems to different applications or different operating systems. Thus, the proposed system needs to be portable and independent on WSN operating systems and applications. Third, the debugging system needs to be adaptable and flexible. As a matter of fact, some bugs might be very easy to be

detected, while some other hidden ones might be difficult to be found. According these different situations, the debugging system should be able to be adapted to go through simple and fast bugs detecting process or a little slower but more powerful bugs detecting process. It will save effort of developers and computing resources.

The WSN debugger is composed of five components as shown in Figure 5.1, a runtime trace collector, a preprocessor, a vPST analyzer, a SVM classifier and a developer or developer group. In the front end, the trace collector generates and collects program runtime traces based on trace points in hot spots of applications. In the back end, the preprocessor, vPST analyzer and SVM classifier work coordinately to detect the most anomalous subsequences in collected traces. The preprocessor filters out noises and uninterested data, compresses the raw data and reduces data dimensionality by preliminary sequential analysis. The vPST analyzer analyzes all the collected runtime traces and extracts the sequential features from traces by vPST. The SVM classifier performs classification based on vPST vectors obtained by the vPST analyzer. Finally developers come to play their roles in the system, they give their feedback back to the system after manually reviewing several very top anomalies in the list. Based on feedback provided by developers, the system iteratively evolves and improves its debugging performance.

In debugging process, a developer initially inserts some trace points wherever he/she thinks the problem might be located according to his/her knowledge of the application source code at that time. Then he/she can run the application in either real deployment or virtual simulations. The only overhead introduced here is collection of a trace during runtime, which is usually very small. After some traces collected, vPST analyzer will analyze the traces and create vPST vectors for each subsequence. Then generated vPST vectors are fed to SVM classifier for anomaly detection. The proposed vPST-SVM debugging system works adaptively and iteratively. Initially SVM just analyzes the root nodes of vPSTs. If bugs can not be detected, the system either takes feedback from developers, removes identified anomalies, iteratively perform more cycles of debugging or simply goes to deeper PST

nodes. The iteration continues until bugs are found and confirmed. By this adaptive debugging procedure, we avoid wasting unnecessary resources.

Overhead for developers of using this debugging system mainly comes from identifying hot spots in the source codes and determining where to insert trace points. Based on our experience of actual WSN development, for most of the cases when a developer encounters some problems with his/her application development, he/she

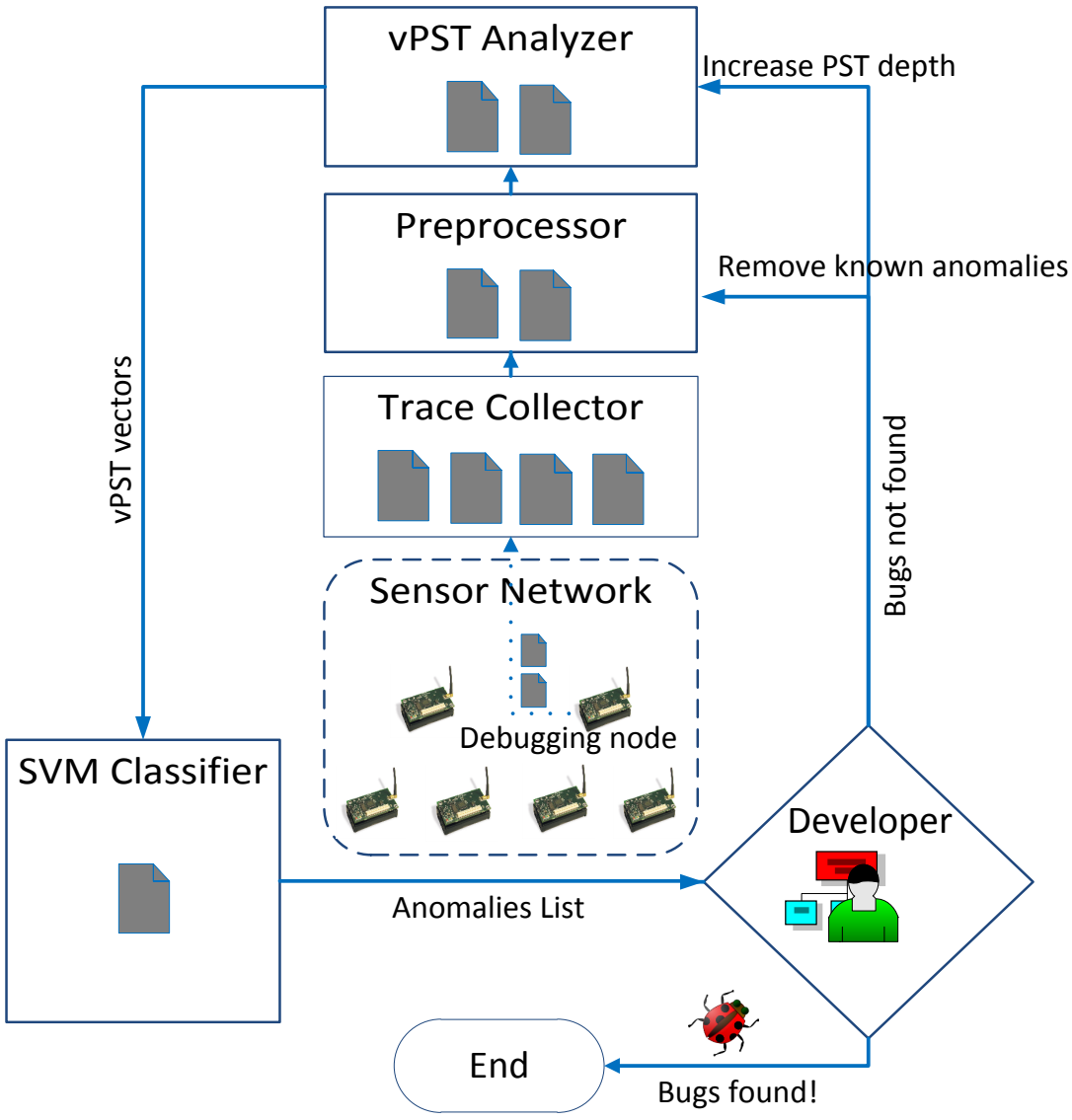

Figure 5.1: System Architecture

intuitively have some sense that some parts of the source code are most probably causing the problem, even he/she is not sure exactly what it is. These problematic blocks of codes are called hot spots. Based on this assumption, I designed the system so that it can be targeting hot spots in the source code. By concentrating on such hot spots, we significantly reduce collected data and simplify problems. Even for the worst cases when the developer is not familiar with the source codes, and does not have prior knowledge of hot spots in the source codes, he/she can still just insert trace points randomly in the source codes. Following preprocessing of the collected traces can identify hot spots composed of trace points with unstable sequential patterns and consolidate those trace points that always form very stable sequential patterns. The reasoning behind this is, hot spots usually generate messy execution traces with many problematic sequential changes while normal code blocks usually generate very stable execution traces without too many changes.

The debugging system is independent on any specific applications or WSN operating systems. Empirically different kinds of WSN applications usually have different types of typical hot spots. For instance, environmental monitoring applications typically have data collection and transmission codes that are prone to contain bugs. So such codes could be potential hot spots in them. For a developer with experience and prior knowledge of the applications, his/her experience is very helpful to reduce overhead of identifying hot spots. If he/she is not experienced of is not familiar with the source codes, there will be some overhead of preprocessing and analyzing runtime traces to locate hot spots. But the overall overhead is significantly reduced by automated trace analyzing.

5.2 System Implementation

5.2.1 Trace Collector

The program runtime trace collector is a front-end data collecting component to collect and store program runtime traces at source code level. Runtime traces are generated by inserting trace points into hot spots in application source codes. For different WSN operating systems, the trace collector need be customized to adapt the corresponding printing service, which is usually a trivial task since most well-developed WSN operating systems have carefully developed printing services and well documented APIs.

In our test case studies, we built lightweight and simple tracing subsystems on both LiteOS and TinyOS. Trace points are simply some distinct numbers that are printed out when at runtime program reaches. The trace points actually compose a simple representation system of program execution sequences. On LiteOS, all the trace points were printed out and sent to desktop computer through serial port via USB cable by using pre-implemented printing functions in libserial.c printing library. On TinyOS, the existing printf library was used to send runtime traces to desktop computer through serial port.

5.2.2 Preprocessor

The sequence preprocessor cleans up collected raw data, compresses those sequences and reduces data dimensionality. Initially, all trace points are inserted into source codes somehow arbitrarily based on the choices of developers. Plenty of information in the collected traces might be useless or redundant. For example, if symbols 1, 2 and 3 always occurs as a whole block “2 1 3” and not any one or two symbols out them occur in different format, we can combine symbols 1, 2 and 3 as a single new symbol and replace the “2 1 3” blocks. By searching and combining these invariant blocks, the dimensionality of original sequences can be reduced. The compression

can improve following sequence analyzing and debugging performance. The sequence preprocessor was implemented in C++.

5.2.3 vPST Analyzer

The main functionality of the sequence analyzer is to extract sequential features from collected data. In principle, there could be huge amount of tracing data collected even after some preprocessing and compressing. It becomes a challenge to extract the most important features from those data. We extended PST to vPST to efficiently extract sequential information from sequential data. We implemented vPST sequence analyzer in C++ for efficiency consideration, since C++ is usually faster than other high level languages, such as Java, Python and etc. The implemented PST construction algorithm followed published algorithm by Ron et al. Ron et al. (1996). PST vectorization followed our description of vPST earlier. The whole vPST sequence analyzer can adaptively extract sequential information from given sequences and store all the information into compact vPST vectors up to given PST depth L.

5.2.4 SVM Classifier

Sequence classifier is a key component of the WSN debugging system. The fundamental principle behind the proposed debugging system is to detect bugs by automatic anomaly detection. Thus, the key functionality of the system is anomaly detection. In other words, the system need be able to classify normal behaviors and abnormal behaviors efficiently. So a robust classifier is necessary for the system to find out the most probable buggy patterns. In recent decades, there are plenty of classification models and algorithms developed for efficient classification, including k-Means, k-nearest-neighbors, SVM and etc. Among these algorithms, SVM is a very robust and generic classifier that can work high-dimensional data, including both linear and nonlinear data. So we chose SVM in the debugging system implementation.

In the debugging system, SVM classifier was built by using a well-developed SVM library LIBSVM Chang and Lin (2011). It was implemented in C++ and took the most advanced computing techniques to achieve its excellent performance. We used both of the Java version and C++ version of it in our experiments. There is no significant difference in their performance.

5.2.5 Developers

Developer is taken into serious consideration in the system design as a system component, because in the proposed debugging system, what computer programs perform are detecting outliers from collected traces, but they can hardly determine which outliers are generated by bugs, and which are from correct but unusual execution. In fact, there might be many outliers generated by normal program execution, but not bugs. Real human being has to take the responsibility of determining whether or not a detected outlier is a bug. Therefore, a developer is an indispensable component of the debugging system. It should be emphasized that adding developer into the system is not to increase his/her burden in development process, but the opposite. Results in test cases will show how a little feedback from developers can significantly improve bugs detecting ability of the debugger in an iterative debugging process.

The roles played by a developer in the debugging system are both a user and a supervisor. First, a developer is a user of the debugger in real WSN application development. Second, a developer is a supervisor and a decision maker of the debugger during the debugging process. Basically, outlier detection algorithms can hardly identify or understand underlying bugs. What the algorithms can obtain is a list of the most problematic runtime subsequences, but not identified bugs. Valuable feedback from real developers is significant to enable the system to learn and improve its prediction performance iteratively. As a result, it is the combination of machine intelligence and human intelligence that makes the debugging system intelligently

evolve to help developers locate real bugs in WSN applications by going through debugging cycles iteratively. The functionality of a developer as a supervisor is to tell the debugger which outliers in the top list are not from bugs and which outliers are from real bugs by manual check of the very short list of top anomalous sequences detected. By such iterative learning, the debugger can achieve much better prediction results in following iterations.

Chapter 6

Evaluations

The proposed debugging system is evaluated by three test cases, one on TinyOS and two on LiteOS. For simple bugs with obvious buggy symptom, like the variable overflow bug in a LiteOS application, a single iteration of vPST-SVM debugging with just a few adjustment of vPST depth can clearly reveal bugs. For those tricky bugs without obvious buggy symptoms, like the race condition bug on TinyOS, several iterations with a little feedback from developers can detect and confirm hidden bugs in runtime traces very well.

6.1 Test Case I: a variable overflow bug in a LiteOS application

In this test case, a LiteOS application that transfers packets between different nodes was developed based on a simple reliable data transfer protocol. Before the bug was found and fixed, we did not have any prior knowledge of any hidden bugs in the application. So the test case studied here was actually a real debugging process of an actual development of a WSN application.

This is basically a simple application that wants to reliably collect data from different senders. In deployment, several sender nodes continuously send radio packets

Table 6.1: Bug 1 trace analyzing results

PST depth	False Alarms	False Positive Rate
0	8	10.26%
1	2	2.56%

to a single receiver node. When received a packet, the receiver always sends back an ACK packet to the sender to notify that the packet has been successfully received. For each sender node, after sending out a packet, it always waits a while for an ACK. If an ACK is not received on time, it just resends the packet. Only after an ACK is successfully received, a new packet will be sent out. There were still a few interesting bugs coming out during the development process of such a simple application. Following debugging process of a variable overflow bug shows how vPST-SVM debugging system could help developers to find and fix WSN bugs efficiently.

The application with the bug looked running normally. No obvious failures. 500 sequences were collected during runtime. vPST-SVM analyzer identified two distinct classes in the trace. One class with repeated patterns, which is "1 2 6 1 3 5 6", started from the 79th subsequence in the trace. This pattern repeated after the application had been running for a little while. Most probably, it was caused by a bug. after further checking this pattern, we found when the bug was triggered, the

```
 1: /* sender node */
 2: while(1)
 3: {...
 4:  if(AckReceived && !MsgSent){
 5:    ...
 6:    lib_radio_send_msg(...);
 7:  }
 8:  else if(!AckReceived && MsgSent){
 9:    lib_radio_receive_timed(...);
10:    // Bug #1
11:    PacketID=256*Buffer[1]+Buffer[0];
12:  }
13: }
```

Listing 6.1: Bug 1 source code

sender would just repeatedly resend the same packet forever. On the receiver, the same packet was always dropped. By checking the source code that generated the corresponding pattern, we finally confirmed and located the bug within pattern "1 3 5 6". The simplified source code is show in Listing 6.1. It was due to a variable overflow. In the code, both PacketID and Buffer[] are defined as 16-bit arrays. But number "256" is a literal number. In compilation, the calculation of 256*Buffer[1] with 16-bit variable and literal number mixed could be compiled in some undefined way. In my experimental setting, when Buffer[1] becomes 3 or larger, the product would be reset to 512. As a result, PacketID would never exceed 512. So the receiver would always think it just has received a very old packet and keep dropping it.

This is a simple bug with obvious symptom in runtime trace, a single iteration of vPST-SVM debugging with simple adjustment of vPST depth to 0 and 1 can differentiate buggy patterns very well. The classification results is shown in Table I. The false positive rate decreased from 10.26% to 2.56% when vPST depth was increased from 0 to 1. There were 8 false alarms in total 500 sequences classified by the vPST-SVM debugging system when vPST depth was set to 0, meanwhile only 2 false alarms when vPST went just one level deeper. The results clearly demonstrated significant improvement could be achieved on prediction performance by taking deeper vPST in the analysis. Deeper vPST means more sequential information was extracted and analyzed by SVM. It shows strong indication that more sequential information included in deeper vPST could help SVM to find hidden bugs more efficiently.

```
 1: /* Thread 1 */
 2: /* receiver node */
 3: while(1)
 4: { lib_sleep_thread(1000);
 5:   if(AckSent && !MsgReceived){
 6:     lib_radio_received_timed(...);
 7:     ...
 8:     fromnodeid = incomingMsg[2];
 9:     if(thisPacketID >= lastPacketID){
10:       lastPacketID = thisPacketID;
11:       AckSent = false;
12:       MsgReceived = true;
13:     }
14:   }
15: }
16: ...
17: /* Thread 2 */
18: /* receiver node */
19: while(1)
20: { lib_sleep_thread(1000);
21:   if(MsgReceived && !AckSent){
22:     ...
23:     // Bug #2.
24:     AckSent = true;
25:     MsgReceived = false;
26:     ...
27:     lib_radio_send_msg(...);
28:   }
29: }
```

Listing 6.2: Bug 2 source code

6.2 Test Case II: a race condition bug in a LiteOS application

In this test case, a WSN bug with a common root cause, race condition, was analyzed by the vPST-SVM debugging system. Similar functionality as the application in test case I was implemented in multi-threaded mode. There are two application threads on each sensor node, one for sending packets, the other for receiving packets. On a sender, one thread is responsible for sending packets, the other one is responsible for receiving ACKs. On the receiver, one thread is responsible for receiving packets sent by senders, the other one is responsible for sending out ACKs to corresponding senders. The two threads share four system state variables as shown in the source code in Listing 6.2, MsgSent, MsgReceived, AckSent and AckReceived, which define finite system states of each node. The application was deployed on 5 different micaZ nodes, 1 receiver and 4 senders. 4 senders were keeping sending messages to the receiver. The receiver sent back ACKs to corresponding senders after it received a message.

A possible failure could be caused by incorrect changing of state variables by one of the two threads. Since both threads have equal access to all system state variables, each thread could change system states by modifying the state variables. In actual execution, the two threads are running in interleaved manner. A failing scenario of the bug studied is as following. Thread B changes system variable AckSent and MsgReceived at line 24 and 25, but before it sends out AckMsg, the thread gives CPU to thread A. When thread A receives a new message from a different sender node and updates fromnodeid on line 8, the bug is triggered. When the CPU is given back to thread B, acknowledgment will be sent to a wrong sender because fromnodeid has been altered. In practical deployment, the bug can be triggered randomly. As a result, buggy subsequences will be sparsely distributed in majority of normal subsequences. It will be difficult to manually check thousands of subsequence to determine if there is something wrong.

Table 6.2: Bug 2 trace analyzing results

Sequence	Anomaly Score	Bug/Anomaly
3033	-4.5467	Anomaly
3070	-4.5467	Anomaly
642	-1.073	Bug
3372	-1.073	Bug
1112	-1.0217	Bug
3144	-1.0217	Bug
380	-1.0002	Bug
...	...	...

In the experiments, 15 total trace points were inserted into source code. Runtime traces were collected and partitioned into subsequences every time thread B ran to a completion. Some inappropriate interleaved execution of thread A and thread B caused bugs. 3400 subsequences in total were collected and analyzed. A single vPST-SVM iteration was conducted with several vPST depth adjustments. Anomaly scores were calculated according to relative positions of data points to found hyperplane. The results ordered by anomaly scores from vPST-SVM analyzing with vPST depth of 5 are shown in Table II. The lower the anomaly score is, the more likely the corresponding subsequence contains bugs. Out of the total 3400 subsequences, there are 5 confirmed bugs within the top 7 anomalies. The first 2 most anomalous subsequences, 3033 and 3070 were not caused by bug but were indeed anomalous sequences that contain the shortest trace points sequence of a thread A cycle and a thread B cycle. Bug they are indeed anomalies compared with majority of other subsequences.

6.3 Test Case III: a race condition bug in a TinyOS application by iterative debugging

The third test case was developed on TinyOS 2.x operating system based on TestFtsp application. In source code of CC2420ReceiveP.nc we added some additional code to

Table 6.3: Bug 3 trace analyzing results

vPST depth	0	1	2	3	4	5
Iteration	Bugs Detected					
First	0	0	0	2	2	1
Second	0	1	2	2	3	5
Third	0	4	3	3	5	5

collect runtime information of the program. As shown in following the pesudo code in Listing 6.3, temporary runtime data of CC2420ReceiveP component is stored in array tmpdata[]. After tmpdata[] is ready with enough data, it will be copied to array tracedata[] and a task sendtrace will be posted to process the collected data. There is a potential race condition bug in this superficially simple application even both of the data collection and processing codes are protected by atomic blocks. At some cases, if the operating system is busy doing something else, before previously collected tracedata is processed in task collecttrace(), new tmpdata may be ready and replaces old data in tracedata. As a results, such missing data can never be collected. In this application, the missing data is very important because it contains program runtime information when the program is busy doing something unexpected, which is crucial in studying detailed program behavior.

```
 1:   task void collecttrace(){
 2:     sendtrace(tracedata);
 3:     printf(" 23\n");
 4:     ...
 5:   }
 6:   ...
 7:   somefunction(){
 8:     tmpdata[] = ...;
 9:     if(tmpdata is ready){
10:       tracedata[]=tmpdata[];
11:       post collecttrace();
12:     }
13:   }
```

Listing 6.3: Bug 3 pesudo code

To uncover this bug, we inserted 23 total trace points into CC2420ReceiveP.nc source code and printed out the trace points using printf library provided by TinyOS 2.x. The whole wireless sensor network for running this test case was built by 9 total micaZ motes. One mote installed RadioCountToLeds application for sending beacon packets over the radio. One mote installed BaseStation for receiving time synchronization information from all synchronizing motes. Among 7 other motes that installed TestFtsp application, one debugging mote installed buggy version of CC2420ReceivedP and was connected to PC computer via a serial cable to collect runtime traces. The trace was broken into smaller pieces at trace point 23 as shown in the pesudo code. Every time trace point 23 was reached, the current subsequence would end and following trace would be printed into a new subsequence. There are two advantages of dividing runtime traces into smaller pieces. First, vPST sequence analyzer can generate vectors from short sequences in reasonable amount of time, while the sequences are too long, it may be unrealistic to conduct vPST-SVM outlier analyzing because it may take too much time. Second, when the most outstanding outliers detected, it is much easier to manually check short subsequences for bugs rather than checking long subsequences. Therefore, partitioning runtime traces into small pieces is for easier inspection of bugs.

This is a tricky bug that is much more difficult to be detected than the bugs in first two test cases. A single iteration of vPST-SVM can hardly detect and confirm the bug. Iterative analyzing was conducted to identify instances of triggered bugs and locate it in the application source code. We started with a collected runtime trace of 2000 total subsequences. At first iteration, 6 experiments on vPST-SVM debugging system were conducted with vPST depth changing from 0 to 5. Very few bugs were identified even vPST depth was set to 5. Then we went to second iteration. First we deleted the top 10 outliers that were identified as non-buggy subsequences from runtime trace file according results from first iteration when vPST depth was 1. Then another 6 vPST-SVM debugging experiments were conducted with vPST depth changing from 0 to 5. Finally a third iteration was performed based on 1980

sequences from second iteration at vPST depth 1 results with another 10 non-buggy outliers deleted. The complete results of all three iterations are shown in Table 3. As shown in the table, The deeper vPST was reached, the more likely bugs could be identified in top 10 anomalies. With more iterations performed, more bugs were detected in top 10 anomalies. It clearly shows deeper vPST can improve debugging performance of the vPST-SVM debugging system. Also it demonstrates the system can improve quickly by learning from developer's input, even only a little feedback was fed to the system.

Chapter 7

Conclusions

In this thesis, I proposed, implemented and evaluated a novel vPST-SVM WSN debugging system by case studies. By applying the debugging system iteratively on collected program runtime traces, hidden bugs in WSN applications can be revealed and identified efficiently. The case studies demonstrated that the vPST-SVM debugging system is a robust, generic, flexible and iterative debugging system that can efficiently learn by inputing a little feedback from developers. For simple bugs with obvious anomalous behavior, a single iteration of vPST-SVM debugging process can identify the bugs very well. For tricky bugs without obvious anomalous symptoms, several iterations of vPST-SVM analyzing by taking feedback from developers can identify bugs reasonably well. By comparison of vPST-SVM performance with different PST depth, I demonstrated that the prediction performance can be improved significantly by taking more sequential information in vPST for SVM classification. In addition, my extension of PST to vPST shed insightful light on future research on sequential data, and my proposed WSN debugging system is the first flexible and iterative debugger for WSN applications that can detect transient bugs very efficiently.

Bibliography

Aizerman, A., Braverman, E. M., and Rozoner, L. I. (1964). Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25:821–837. 2, 12

Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm. 25

Christine, L. (2003). Prediction suffix trees for supervised classification of sequences. *Pattern Recognition Letters*, 24(16):3153–3164. 10

Jiang, L. and Su, Z. (2007). Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 184–193, New York, NY, USA. ACM. 2

Keerthi, S. S., Chapelle, O., and DeCoste, D. (2006). Building support vector machines with reduced classifier complexity. *Journal of Machine Learning Research*, 7:1493–1515. 18

Khan, M. M. H., Le, H. K., Ahmadi, H., Abdelzaher, T. F., and Han, J. (2008). Dustminer: troubleshooting interactive complexity bugs in sensor networks. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 99–112, New York, NY, USA. ACM. 6

Kulakov, A. and Davcev, D. (2005). Tracking of unusual events in wireless sensor networks based on artificial neural-networks algorithms. In *International*

Conference on Information Technology: Coding and Computing, 2005. ITCC 2005, volume 2, pages 534–539. IEEE. 1

Kwon, J., Chen, C., and Varaiya, P. (2004). Statistical methods for detecting spatial configuration errors in traffic surveillance sensors. *Transportation Research Record*, 1870(1):124–132. 1

Langendoen, K., Baggio, A., and Visser, O. (2006). Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE. 1

Leonardi, F. G. (2006). A generalization of the PST algorithm: modeling the sparse nature of protein sequences. *Bioinformatics*, 22(11):1302–1307. 10

Li, P. and Regehr, J. (2010). T-check: bug finding for sensor networks. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 174–185, New York, NY, USA. ACM. 4, 5, 6

Liao, L. and Noble, W. S. (2003). Combining pairwise sequence similarity and support vector machines for detecting remote protein evolutionary and structural relationships. *Journal of Computational Biology*, 10:857–868. 10

Liu, H., Selavo, L., and Stankovic, J. (2007). Seedtv: deployment-time validation for wireless sensor networks. In *Proceedings of the 4th workshop on Embedded networked sensors*, EmNets '07, pages 23–27, New York, NY, USA. ACM. 4

Lu, K., Cao, Q., and Thomason, M. (2012). Bugs or anomalies? sequence mining based debugging in wireless sensor networks. *Mobile Ad-Hoc and Sensor Systems, IEEE International Conference on*, 0:463–467. 2

Mazeroff, G., Gregor, J., Thomason, M., and Ford, R. (2008). Probabilistic suffix models for API sequence analysis of windows XP applications. *Pattern Recogn.*, 41(1):90–101. 10, 11, 18

Ramanathan, N., Kohler, E., and Estrin, D. (2005). Towards a debugging system for sensor networks. *International Journal of Network Management*, 15(4):223–234. 1, 2

Ron, D., Singer, Y., and TIshby, N. (1996). The power of amnesia: Learning probabilistic automata with variable memory length. *Machine Learning*, 25:117–149. 2, 9, 10, 11, 24

Sasnauskas, R., Landsiedel, O., Alizai, M. H., Weise, C., Kowalewski, S., and Wehrle, K. (2010). Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 186–196, New York, NY, USA. ACM. 4, 6

Schoofs, A., O"Hare, G., and Ruzzelli, A. (2012). Debugging low-power and lossy wireless networks: A survey. *Communications Surveys Tutorials, IEEE*, 14(2):311–321. 4

Schlkopf, B., Platt, J. C., Shawe-Taylor, J. C., Smola, A. J., and Williamson, R. C. (2001). Estimating the support of a High-Dimensional distribution. *Neural Comput.*, 13(7):1443–1471. 12

Sreedevi, T. and Sebastian, M. (2012). A classification of the debugging techniques of wireless sensor networks. In *2012 International Conference on Advances in Computing and Communications (ICACC)*, pages 51–57. 4

Sun, P., Chawla, S., Arunasalam, B., Ghosh, J., Lambert, D., Skillicorn, D., and Srivastava, J. (2006). Mining for outliers in sequential databases. In *SDM*. SIAM. 14

Vapnik, V. N. (1995). *The nature of statistical learning theory*. Springer-Verlag New York, Inc., New York, NY, USA. 9

Werner-Allen, G., Lorincz, K., Johnson, J., Lees, J., and Welsh, M. (2006). Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 381–396, Berkeley, CA, USA. USENIX Association. 1

Zhou, Y., Chen, X., Lyu, M. R., and Liu, J. (2010). Sentomist: Unveiling transient sensor network bugs via symptom mining. In *2010 IEEE 30th International Conference on Distributed Computing Systems (ICDCS)*, pages 784–794. IEEE. 4, 5, 7

Appendix

Vita

Kefa Lu was born in Anqing, Anhui, China in 1983. He earned his Bachelor of Science in Physics from Department of Physics in University of Science of Technology of China in Hefei, Anhui, China, in 2004. Later in 2010 he earned his Master of Science in Chemistry from University of Utah in Salt Lake City, Utah, United States. He worked as a research assistant. His research areas are computational chemistry, computational biochemistry and computational biophysics. In fall of 2010, he enrolled into doctoral program at the University of Tennessee at Knoxville in the department of Electrical Engineering and Computer Science. At the same time, he joined the Laboratory for Autonomous Interconnected, and Embedded Systems (Lanterns) group as a graduate teaching assistant and research assistant. Later 2012 he changed to Masters program, and will complete his Master's Degree in Spring of 2013. His major research areas are wireless sensor networks, distributed systems, data mining and machine learning.