Post-Deployment Anomaly Detection and Diagnosis in Networked Embedded Systems by Program Profiling and Symptom Mining

Wei Dong, Member, IEEE, Luyao Luo, Chun Chen, Member, IEEE, Jiajun Bu, Member, IEEE, Xue Liu, Member, IEEE, Yunhao Liu, Fellow, IEEE

Abstract—Detecting and diagnosing anomalies in networked embedded systems like sensor networks is a very difficult task, due to the variable workloads and severe resource constraints. In this paper, we focus on how to aid bug diagnosis after the system has been deployed. We notice that most node-level debugging tools can provide detailed program information inside the node but fail to detect when and where a problem occurs in the network. On the other hand, most network-level diagnosis tools can effectively detect a problem from the network but fail to narrow down the problem within the node because they lack detailed program information. To close the gap, we propose D2, a new method for post-deployment anomaly detection and diagnosis in networked embedded systems by combining program profiling and symptom mining. D2 employs binary instrumentation to perform lightweight function count profiling. Based on the statistics, D2 uses PCA (Principal Component Analysis) based approach for automatically detecting network anomalies. Compared with previous methods, D2 is able to point programmers closer to the most likely causes by a novel approach combining statistical tests and program call graph analysis. We implement our method based on TinyOS 2.1.1 and evaluate its effectiveness by case studies in the development of a working sensor network. Results show that our method can aid programmers to diagnose problems quickly in real-world sensor network systems, and at the same time, incurs an acceptable overhead to the running system.

Index Terms—Networked embedded systems; sensor networks; diagnosis; program profiling; symptom mining

1 INTRODUCTION

Detecting and diagnosing anomalies in networked embedded systems like sensor networks is a very difficult task, due to the variable workloads and severe resource constraints. Many real-world deployments exemplify such difficulties.

LOFAR-agro is a sensor network consisting of about 100 nodes for precision agriculture in the Netherlands during the year 2004-2005 [1]. The software components include (1) TMAC, an adaptive low-duty-cycle MAC protocol, (2) MintRoute [2], a multihop routing protocol, (3) Deluge [3], a wireless reprogramming protocol, and (4) the LOFAR-agro application. The developers encounter numerous problems during the deployment. They observe that the system exhibits low data rate due to the malfunction of TMAC. Detailed diagnosis requires a more than thorough understanding of the TinyOS structure and its components. The concrete causes are thus left unclear due to tight project schedules.

GreenOrbs is a large-scale sensor network system consisting of over 300 nodes for forestry applications starting from the year 2004-2005 [4]. The initial software components include (1) CTP [5], a multihop routing protocol, (2) FTSP [6], a global time synchronization protocol, and (3) the GreenOrbs application. During the deployment in the Zhejiang Forestry University’s woodland, we often observe that some nodes frequently lose synchronization with the rest of the network. After many rounds of careful detections involving both code reviews and testbed experiments, we eventually find out the causes. First, the TinyOS clock driver would occasionally return bogus local timestamps. Second, FTSP does not check the validity of the timestamps.

These experiences show us the importance of problem detection and diagnosis as well as their practical challenges in real-world systems. There are numerous methods to aid problem diagnosis in the literature, such as static verification [7], [8], interactive debugging [9], [10], tracing and logging [11], [12], [13], network-level diagnosis [14], [15], [16], etc. In this paper, we focus on functional problems after the system is deployed. Simulation and testbed environments are inevitably different from real deployments. Some problems can only manifest themselves after deployments. Functional problems should be detected and diagnosed in time since their occurrences at some nodes usually suppress their normal work.

We notice that most node-level debugging tools can provide detailed program information inside the node but may fail to detect when and where a problem occurs in the network. On the other hand, most network-level diagnosis tools can effectively detect a problem from the network but may fail to narrow down the problem within the node because they lack detailed program information. A simple combination of the above two will cause large overhead. Moreover, some errors detected by network-level tools may not be reproducible and thus cannot be easily diagnosed by the node-level tools.

To close the gap, we propose D2, a post-deployment anoma-
D2 combines statistical tests and program call graph analysis to point programmers closer to the most likely causes by a novel approach combining statistical tests and program call graph analysis.

It is worth noting some of the most important features in D2.

- D2 does not require efforts from the developers. The source code does not need to be modified and no libraries need to be linked into the executable file. Therefore, it is easily scalable to a wide range of applications.
- D2 does not affect program execution unless the sink node issues a profiling request. D2 can be installed or uninstalled easily at the runtime so that we can use it when needed and stop it otherwise.
- D2 uses lightweight function count profiling. Counter-based profiling incurs less overhead than event-based profiling.
- Despite lightweight, D2 provides function-level information. Note that functions for sensor networks are typically very short. Hence function counts for sensor network applications carry more execution information than those for PCs.
- D2 automates the process of problem identification and causal reasoning. The diagnosis report provides detailed information for problem solving.
- Although we currently implement D2 based on TinyOS, its design principles can also be applied to other OSes since binary instrumentation within D2 operates at the binary level.

We evaluate D2’s effectiveness by case studies in real-world sensor network applications. Results show that our method incurs an acceptable overhead and can aid programmers to diagnose real-world problems quickly.

We specialize in detecting and diagnosing functional problems after deployment, which leads to two important design considerations. First, the tool should be lightweight so that normal operations are not severely affected. Second, the profiling of function executions is needed for fine-grained diagnosis inside the node.

It is also important to mention the limitations of our tool. First, in order to correctly detect an anomaly, system data (i.e., program profile) about the anomaly needs to be collected since D2 relies on the collected system data for problem detection and diagnosis. If a non-producible anomaly happens before D2 is turned on, this anomaly is left undetected. Second, D2’s function count profiling approach, albeit lightweight, loses other detailed function execution information, e.g., the return value of each function. D2 can be extended to incorporate more advanced profiling methods such as call site profiling with return values [19], at the cost of large profiling overhead. Third, D2 cannot detect concurrency problems due to improper interleavings of functions. Detecting such bugs requires detailed function-level logging to track the order of function executions using tools such as TinyTracer [11] or LIS [20], [21].

The contributions of this work are summarized as follows.

- We propose a new method for post-deployment anomaly detection and diagnosis in networked embedded systems by combining program profiling and symptom mining.
- We propose a novel approach combining statistical tests and program call graph analysis to point programmers closer to the most likely causes inside the node.
- We implement our method and demonstrate its effectiveness using case studies from real sensor network applications.

The rest of this paper is structured as follows. Section 2 describes the related work. Section 3 provides the preliminaries. Section 4 presents the design principles. Section 5 describes D2’s extensions. Section 6 shows the evaluation results, and finally, Section 7 concludes this paper and gives directions of future work.

## 2 Related Work

There are numerous research works related to anomaly detection and diagnosis. Table 1 summarizes diagnosis tools for networked embedded systems. We classify existing works into four main categories: pre-deployment tools, debugging tools, logging and tracing, network-level diagnosis.

<table>
<thead>
<tr>
<th>Tools</th>
<th>Post-deployment diagnosis ability</th>
<th>Automatic detection</th>
<th>Code-level diagnosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-deployment tools</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-Check</td>
<td>×</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>KleeNet</td>
<td>×</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>T-Morph</td>
<td>×</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Semtonist</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Debugging tools</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clairvoyant</td>
<td>√</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>NodeMD</td>
<td>√</td>
<td>×</td>
<td>√</td>
</tr>
<tr>
<td>DT</td>
<td>√</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Logging and tracing tools</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EnviroLog</td>
<td>√</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>DunMiner</td>
<td>×</td>
<td>√</td>
<td>×</td>
</tr>
<tr>
<td>TinyTracer</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>LIS</td>
<td>√</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Aveksha</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Network-level diagnosis tools</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sympathy</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>PAD</td>
<td>√</td>
<td>√</td>
<td>×</td>
</tr>
<tr>
<td>AD</td>
<td>√</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>TinyD2</td>
<td>√</td>
<td>√</td>
<td>×</td>
</tr>
<tr>
<td>LD2</td>
<td>√</td>
<td>√</td>
<td>×</td>
</tr>
<tr>
<td>This paper</td>
<td>D2</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

We specialize in detecting and diagnosing functional problems after deployment, which leads to two important design considerations. First, the tool should be lightweight so that normal operations are not severely affected. Second, the profiling of function executions is needed for fine-grained diagnosis inside the node.

It is also important to mention the limitations of our tool. First, in order to correctly detect an anomaly, system data (i.e., program profile) about the anomaly needs to be collected since D2 relies on the collected system data for problem detection and diagnosis. If a non-producible anomaly happens before D2 is turned on, this anomaly is left undetected. Second, D2’s function count profiling approach, albeit lightweight, loses other detailed function execution information, e.g., the return value of each function. D2 can be extended to incorporate more advanced profiling methods such as call site profiling with return values [19], at the cost of large profiling overhead. Third, D2 cannot detect concurrency problems due to improper interleavings of functions. Detecting such bugs requires detailed function-level logging to track the order of function executions using tools such as TinyTracer [11] or LIS [20], [21].

The contributions of this work are summarized as follows.

- We propose a new method for post-deployment anomaly detection and diagnosis in networked embedded systems by combining program profiling and symptom mining.
- We propose a novel approach combining statistical tests and program call graph analysis to point programmers closer to the most likely causes inside the node.
- We implement our method and demonstrate its effectiveness using case studies from real sensor network applications.

The rest of this paper is structured as follows. Section 2 describes the related work. Section 3 provides the preliminaries. Section 4 presents the design principles. Section 5 describes D2’s extensions. Section 6 shows the evaluation results, and finally, Section 7 concludes this paper and gives directions of future work.

## 2 Related Work

There are numerous research works related to anomaly detection and diagnosis. Table 1 summarizes diagnosis tools for networked embedded systems. We classify existing works into four main categories: pre-deployment tools, debugging tools, logging and tracing, network-level diagnosis.

These diagnosis tools provide the following features:
• **Post-deployment diagnosis ability.** For post-deployment diagnosis, the tool should incur small runtime overhead. Moreover, it should be installed or uninstalled easily at the runtime so that we can use it when needed and stop it otherwise. All the mentioned pre-deployment tools lack post-deployment diagnosis ability since they require fine-grained information that is only affordable by simulators (e.g., Sentomist [22], T-Morph [23]). Some of the tools (e.g., DustMiner [24] and LIS [20]) cannot be uninstalled easily: once installed, they affect the program’s execution and incur runtime overhead to the system.

• **Automatic detection.** Automatic anomaly detection is very important. Most debugging and logging tools (except DustMiner [24]) lack automatic detection ability. Instead, they require developers’ experiences to record or probe the most appropriate program information.

• **Code-level diagnosis.** Code-level diagnosis is very useful for developers. In particular, descriptive diagnosis reports are desired to fix the program problems. All the mentioned network-level diagnosis tools can only pinpoint the problematic nodes or links.

Table 1 shows that none of these tools achieve all these features simultaneously.

**Pre-deployment tools.** T-Check [7] is a tool that uses random walks and explicit state model checking to find safety and liveness errors in sensor network applications running on TinyOS. T-Check is based on the TOSSIM simulator and thus loses the ability to detect and diagnose real-world sensor network software. KleeNet [8] uses symbolic analysis to generate test cases for sensor network code. T-Morph [23] is a novel tool to mine, visualize, and verify the execution patterns of TinyOS applications. T-Morph abstracts the dynamic execution process of a TinyOS application into simple, structured application behavior models, which well reflect how the static source codes are executed. Sentomist [22] uses the number of executed instructions during interrupt handling intervals to find transient bugs.

Both T-Morph and Sentomist are based on Avrora [25]—an instruction level simulator for the mica platform. Such detailed information can only be acquired in simulations and is not affordable in real-world sensor network deployments. Unlike Sentomist, D2 further generates diagnosis reports to facilitate problem diagnosis.

**Debugging tools.** Clairvoyant [9] is a comprehensive source-level debugger for wireless embedded networks. With Clairvoyant, a developer can execute GDB-like commands to interactively debug the sensor nodes. NodeMD [26] is designed to diagnose node-level faults in sensor network applications. It focuses on catching software faults before they completely disable the remote sensor node, so that the user can be provided with diagnostic information to troubleshoot the root cause. Declarative Tracepoint (DT) [10] integrates benefits of previous debugging techniques and uses a SQL-based language interface for debugging.

Although these tools facilitate fixing the already-seen bugs, they cannot automatically identify such bugs.

**Logging and tracing.** EnviroLog [27] aims to improve repeatability of experimental testing of distributed event-driven applications, based on the observation that the system state can change depending on the event sequence and timing. EnviroLog provides an event recording and replay service that captures and replays events with the help of the nonvolatile flash. DustMiner [24] identifies bugs in sensor network software by checking discriminative log patterns. Sundaram et al. propose TinyTracer, an efficient intra-procedural and inter-procedural control-flow tracing algorithm that generates the traces of all interleaving concurrent events [11]. AVEKSHA [12] is a hardware-software approach for tracing events in a non-intrusive manner. LIS (Log Instrumentation Specification) [20], [21] performs insertion of low overhead logging calls into a system. At the heart of LIS are the three scoping declarations (i.e., global, local, point) that direct the assignment of the token identifier that is logged by the system. By separating the local and point scope from the global scope, LIS can significantly reduce the number of bits required to log a given token.

Compared with some approaches with manual logging, D2 uses automatic function count profiling. Although it is possible that manual logging can detect finer-grained program problems (e.g., concurrency problems) by logging important events or variables, it highly depends on developers’ experiences. For the Internet or high-performance distributed systems, the logging information is typically available [28]. However, for most sensor network software, the logging information is usually unavailable since it is often not a good idea for detailed logging which incurs a large runtime overhead. D2’s counter-based profiling can be more efficient than LIS’s event-based profiling and we will make a quantitative comparison in Section 6.

**Network-level diagnosis.** Sympathy [14] collects multiple network metrics and uses a decision tree to localize the failures. PAD [15] uses lightweight network monitoring and Bayesian network based analysis to infer network failures and their causes. Agnostic Diagnosis [17] also collects multiple network metrics and uses anomaly detection on the correlation graph to discover silent failures. TinyD2 [16] uses the concept of self-diagnosis in which each sensor can join the fault decision process. TinyD2 [16] plants a finite state machine into each sensor node, enabling them to accurately change the diagnosis state. LD2 [18] is a in-network diagnosis approach which conducts the diagnosis process in a local area. LD2 achieves diagnosis decision through distributed evidence fusion operations.

Although these tools can automatically narrow down the problem to the node level or link level, they cannot localize the problem inside the node, e.g., the program code. Moreover, they demand manual efforts to write customized node-level diagnosis engine or additional codes for obtaining network metrics.

### 3 Preliminaries

This section introduces the necessary techniques. The trampoline technique is used in the binary instrumentation phase in order to perform the actual profiling task without significantly modifying the original code. One-class SVM (Support Vector
Machine) and PCA (Principal Component Analysis) are two well-known methods for anomaly detection. D2 uses PCA for anomaly detection.

The trampoline technique. In embedded systems, trampolines are short snippets of code that start up other snippets of code. A trampoline is inserted into the target binary, such that the target execution is rerouted to the patch, before it returns for execution of the original code [29]. In our case, we use the trampoline technique to redirect the control to a target function which performs the actual profiling tasks.

SVM, one-class SVM and PCA. SVM is a statistical method that infers how two classes of points are different from each other. Given a set of labeled data points, the algorithm can find a hyperplane that best separates a set of unlabeled data points into the two different classes. The hyperplane is considered as the boundary of the two classes.

When it is used for anomaly detection, what we have is a set of unlabeled ones. A trick is therefore to assume that all input samples belong to one class, i.e., the normal class, which however contains some misclassified ones. Also consider that there is a virtual outlier class, which naturally contains the origin in a $d$-dimensional space and some samples that are misclassified to the normal class. We can then apply SVM to find a boundary to separate these two classes. Such a variant of SVM is called one-class SVM [22].

PCA is a way of identifying patterns in data, and expressing the data in such a way as to highlight their similarities and differences. PCA is a powerful tool for analyzing data of high dimension and has been applied in many areas. PCA captures patterns in high-dimensional data by automatically choosing a set of principal components (i.e., coordinates). When it is used for anomaly detection, we consider the principal components form the normal subspace. The distance from a data point to the normal subspace can be used for anomaly detection. A simple heuristic is that the farther it is away from the normal subspace, the more suspicious it is as an outlier. The runtime overhead of PCA is linear with the number of feature vectors and thus can scale to large data.

Both one-class SVM and PCA can model the majority characteristics of a set of unclassified samples and determine whether a sample is an outlier. D2 uses PCA for anomaly detection. We will explain PCA is a more suitable choice in our case in Section 4.3.

4 Design

Figure 1 shows an overview of D2. The sink node can issue a request to notify a subnet of nodes to transition into profiling mode. Once requested, the D2 module on the sensor node employs binary instrumentation to perform function count profiling. Snapshots of the profiles are either transferred to the sink for real-time analysis or stored on the sensor nodes’ external flash for later analysis.

The D2 module at the PC side performs analysis on the collected profiles. First, D2 performs PCA-based anomaly detection to identify when and where potential problems occur in the network. Second, D2 tries to narrow down the problem to function level by considering statistical divergence of functions or function ratios between normal profiles and abnormal profiles. The diagnosis report lists a set of suspicious functions or function ratios. The result is further refined by their causal relationships which are obtained by inspecting the function call graph of the program.

4.1 Binary instrumentation

The D2 module on the sensor node is responsible for instrumenting the program binary to perform function count profiling. The D2 module employs binary instrumentation technique which inserts additional code and data into the executable, modifying the runtime behaviors to perform the needed task [30]. In addition, D2 maintains necessary information to undo the profiling task.

Function count profiling needs to find the start of each function. The D2 module performs a simple disassembly of the program, discovering every function block by examining the destination of every call instruction in the code. This approach will not reveal functions that are called only by function pointers. However, such functions are not common in TinyOS, and if they must be profiled then the symbol table generated from a compiler can be loaded.

Function count profiling also requires a set of counters to be allocated in order to count the number of each function’s executions. The RAM on current sensor nodes are divided into three sections as shown in the left part in Figure 2. The program’s initialized data, .data, is allocated at the start of the RAM. The program’s uninitialized data, .bss, is allocated following the .data. The program’s execution stack grows from the bottom of the RAM. D2 tries to allocate these counters after the .bss section so that the program’s data would not be corrupted and the stack space can be maximized. On PCs, it is easy to find out the end of .bss by looking at the value of __end__bss in the ELF file [31], indicating the end of the .bss section. On the sensor nodes, however, only the raw binary file is stored and detailed metadata information.
is lacking. D2 finds out the end of .bss by examining the initialization procedure whose program logic is common for almost all applications. D2 obtains the end of .bss section by examining the corresponding constants encoded in the instructions. D2 allocates 4 bytes for each function counter. For a complex application like GreenOrbs with about 280 functions, the overall RAM overhead is 1,120 bytes. This is acceptable compared with a total of 10KB RAM on TelosB nodes.

D2 uses the trampoline technique to replace the instruction block at the start of each function by a call instruction so as to direct the control to the trampoline which performs the actual profiling.

Figure 3 shows the basic idea of the trampoline technique. We explain each step in the instrumentation process as follows.

1) At the start of each function, we replace the original instructions to a call instruction which directs the control to the corresponding trampoline. A call instruction may replace one or two instructions as the instructions are of variable lengths. The replaced instructions are “mirrored” at the corresponding trampoline and meant to be executed after function count profiling is finished.

2) The call instruction directs the function’s control to its trampoline. The trampoline first saves the context by saving values of registers that will be used by the trampoline. This is achieved by pushing the values onto the stack.

3) The trampoline executes the profiling logic. First, the corresponding function counter is incremented. Second, a checking procedure is executed. The main task of the checking procedure is to check whether it is the right time to take a snapshot of the function counters by transferring them to the sink or storing them on the local external flash.

4) The context is restored by popping values from the stack to corresponding registers.

5) The mirrored instructions in the original function are executed. It is worth mentioning that if we have to replace relative instructions, we cannot simply mirror the instructions at a different location. Instead, we should translate the instructions to use the absolute address.

6) The control is finally transferred to the function.

We have mentioned that a checking procedure needs to be invoked in order to take snapshots of the function counters. These snapshots are used to create features over a time window for problem detection. For traditional PC software, snapshots may not be needed [19] because the execution of most PC software (such as latex, gcc) finishes in a short time. On the other hand, sensor software is executed for a long duration. Program features over a relatively short time window will enable problem detection at a fine-grained time granularity.

### 4.2 Problem detection

A key problem is how to determine when we should take snapshots. A naive approach would take snapshots at a predetermined time window. This approach will cause extra overhead if no activities happen during the time window. It is not uncommon that sensor nodes perform tasks periodically and infrequently. Hence, it is important to reduce the snapshot overhead when sensor node remains in the sleep state.

To address this issue, D2 adaptively takes snapshots. The checking procedure checks a total function counter (which counts the total number of function executions) at the current time as well as at the last snapshot. If it detects that the difference of the current total function counter and the total function counter at the last snapshot exceeds a threshold, e.g., 5000, it takes snapshots of the function counters by either transferring snapshots to the sink or saving snapshots to the external flash.

We note that the setting of this threshold has a tradeoff. On one hand, if the threshold is too large, the time granularity may not be fine-grained enough for detecting transient problems. On the other hand, if the threshold is too small, the snapshot overhead will be large. The threshold can also be dynamically reconfigured using a data dissemination protocol such as Drip [32] or DIP [33].
detect anomalies among the given function count vectors. For the example shown in Figure 4, the second vector may be identified as an anomaly since the ratio of send() and receive() is much smaller than those in most other vectors.

A key assumption of our approach is that a problem makes a sensor node deviate from the normal, and thus outliers are good indicators of potential problems. During normal executions the relative frequency of two function counts in a time window usually stays the same. For example, the ratio between functions send() and receive() in the CTP component [5] is usually very stable during normal executions, but changes significantly when a problem occurs. The actual count does not matter (as it depends on workloads), but the ratio among different function counts matters.

We would like to “find needle from the haystack”, i.e., detect anomalies from all the diagnostic data. We adopt the Principal Component Analysis (PCA) approach. PCA is able to capture the essence of correlation in the data. Figure 4 illustrates an example using two dimensions in our data, i.e., function counts of send() and receive(). We see that most data resides in the straight line of $S_n$. The axis $S_n$ captures the strong correlation between the two dimensions. Therefore, data points far from $S_n$, e.g., B and C, show unusual correlation, and thus are considered as anomalies. Point C represents that the number of send() is much smaller than the number of receive(), indicating that the node may experience an overflow in the receiving queue. Point B represents that the number of receive() is much smaller than the number of send(), indicating that the node may experience a high number of retransmissions. On the other hand, data point A, though far from other points, is close to $S_n$, and thus is considered as normal. Point A represents cases in which both send() and receive() are executed frequently, indicating that the node has a large workload during that time window.

It is important to note that the problematic function can be easily identified for some simple cases. But problem detection becomes much more difficult for complex cases. For example, when the correlation of a pair of functions changes from high to low, or when additional system information (e.g., timing of I/O operations) are considered. PCA can not only work for simple cases but also for complex cases, making our approach easily extensible. It is also important to contrast PCA with one-class SVM for anomaly detection. For a similar set of data points, we apply one-class SVM and Figure 6 shows the detection results with the gray scales representing the distances to the hyperplane. We can see that PCA identifies B and C as anomalies while SVM identifies B, C, D, E as anomalies (if the threshold is set near the outmost boundary). In reality, data points D and E may represent normal conditions where the node experiences heavy workload for forwarding data packets. Therefore, we consider PCA as a more suitable choice in our case.

Like [28], we use the distance from a data point to the normal subspace $S_n$ to determine whether $f$ is an anomaly. The Squared Prediction Error $SPE = ||f_a||^2$ where $f_a = (I - PP^T)f$ is the projection of $f$ onto the abnormal subspace $S_a$, and $P = [p_1, p_2, ..., p_k]$ where $p_1, p_2, ..., p_k$ are the principal components.

Fig. 5: PCA anomaly detection. $S_n$ indicates the normal subspace and $S_a$ indicates the abnormal subspace. $f$ represents the feature vector. $f_a$ is the projection of $f$ onto the abnormal subspace $S_a$. SPE (Squared Prediction Error) is the squared length of $f_a$. Points B and C are identified as anomalies.

Fig. 6: Anomaly detection using one-class SVM. The gray scales represent the distances to the hyperplane.

We use a threshold of $Q_\alpha$ to detect whether a data point is abnormal:

$$SPE = ||f_a||^2 > Q_\alpha$$

(1)

where $Q_\alpha$ denotes the $Q$ statistic (a well known test statistic for the SPE residual function [34]) at the $1 - \alpha$ confidence level. The choice of the confidence parameter $\alpha$ for anomaly detection has been studied in previous work [35]. We choose $\alpha = 0.001$ as in previous work [28].

4.3 Problem diagnosis

Now we have detected abnormal function count vectors. We still lack detailed information why the anomaly occurs. For the example shown in Figure 4, the second vector may be detected as an anomaly. But we still do not know which specific functions contribute to this abnormal behavior. For a
practical system with hundreds of functions, it is very useful to narrow down to the problematic functions.

Generally, given an anomaly and a set of normal profiles, we would like to examine each individual dimension and the ratios between two dimensions to see whether they are significantly different from the normal data. For example, we have detected that a node exhibits abnormal behaviors with its feature vector [11,100,22,38] in which each element corresponds to each function execution counter. We have also determined that the normal feature vector should be [10±2.60±50,20±3.30±4] where each element encodes both a mean value and a standard deviation. We would like to determine which function or a pair of functions are abnormal from the normal data. Considering that a program may contain hundreds of different functions in practice, it is useful to narrow down to the problematic functions for bug fixing. For this particular case, we can detect that the last function is abnormal since its execution counter far exceeds the normal data while the first three function counters are all within the normal range. D2 not only detects single abnormal function, but also abnormal function execution ratios. It performs such detections in a n² matrix with each element corresponding to a ratio for a pair of functions.

We perform t-tests [36] to compare n function counts and n² ratios between the detected anomaly and the normal data points. If the t-test rejects the null hypothesis, we conclude that the corresponding function count or ratio is able to distinguish the two classes. We use Welsh’s t-test and use a critical value of p < 0.05 to reject the null hypothesis and assess significance.

The magnitude of the t-statistic indicates the difference between the two classes. A larger t-statistic can be due to a larger difference in the means and/or smaller variance in the two classes. The sign of the t-statistic indicates which class has a bigger mean [37].

At this time, we can return a list of suspicious functions or ratios between two functions ranked by their statistical significance. The result can further be refined by considering the call/post relationship between functions. For example, we have ranked functions A, B, C at the top. Without their relationships, we need to manually check the correctness of all these functions. Clearly, a diagnosis report showing both statistical difference as well as the call/post relationships can greatly help further diagnosis.

We define elements in the diagnosis report as follows.

- A node represents a suspicious function or a suspicious ratio between two functions.
- The size of the node indicates the statistical difference from the normal data.
- There is a directed edge from node A to node B if functions in node A are predecessors of functions in node B in the call/post graph.
- If some suspicious functions share common ancestors, we also show the nearest common ancestor and the corresponding call/post relationship to facilitate problem reasoning.

Figures 9–12 show examples of diagnosis reports. In the example diagnosis report shown in Fig. 12, we see that three function counts decrease rapidly at the problematic node. All these functions are related to the CTP sending logic. This localizes the bug to the corresponding component. The decrease of these function counters implies that the problematic node stops invoking sendMessaget() due to some reasons in the radio stack. This gives programmers further guidance on how to diagnose the bug by inspecting the sending-related functions in the corresponding component.

We obtain the call/post graph of the program by parsing the ELF file on the PC. The call relationship is parsed by examining the call instructions and the corresponding target addresses in the code section. As mentioned earlier, call with pointers is a rare condition in TinyOS.

The post relationship is obtained by a more complicated way in the code section. The execution model of TinyOS consists of interrupts and tasks. Interrupts execute at a higher priority and can preempt the execution of tasks. Tasks execute at a lower priority and are scheduled in a FIFO manner. Interrupts are used to handle time sensitive operations which are usually very short. Tasks can be posted in the interrupts or other tasks to continue the processing a complex logical task. In TinyOS, the postTask function is used to post a task to the FIFO task queue. The runTask function is used to schedule the execution of queued tasks when the TinyOS task scheduler gains the CPU. We use the following procedure to obtain the post relationship. First, the task ID is found by inspecting calls to TinyOS postTask. Second, the functions corresponding to the actual task is found by inspecting the switch...case table in the runTask function with the task ID.

5 Extension

We have described how D2 can detect and diagnosis anomalies exhibited in the function count vectors. In some circumstances, it is important to consider the timing information of program executions. For example, Distalyzer [37] extracts two features from a large amount of log data in distributed systems. The event feature summarizes the timing of system events while the state feature summarizes the values of a set of state variables.

A native approach to extend D2 is to additionally record the execution time of a function in the binary instrumentation process. However, we find that the execution time of a function in TinyOS is not informative enough. This is because I/O operations (e.g., send a packet, read a sensor data, write to the external flash, etc) are heavily used in event-driven systems like TinyOS, and, the I/O waiting time cannot captured in the function execution time since I/O operations in TinyOS are in split phase.

We extend D2’s binary instrumentation process to record the waiting time of the split-phase I/O operations, e.g., the time interval between send() and sendDone(), or, read() and readDone(). Therefore, the snapshots also include the average I/O waiting time of the split-phase I/O operations. Hence, we can additionally get I/O timing vector f which records I/O operation’s average waiting time. The PCA anomaly detection process detects anomalies based on both feature vectors f (function count) and I (I/O timing). By this approach, we can not only detect statistical anomalies exhibited in function counts but also anomalies exhibited in I/O waiting times.
Table 2: The number of principal components in the feature data. \( n \) is the dimension of the feature vector \( f \) and \( k \) is the number of principal components.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>( n )</th>
<th>( k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestNetwork</td>
<td>257</td>
<td>10</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>212</td>
<td>14</td>
</tr>
<tr>
<td>TestDip</td>
<td>148</td>
<td>13</td>
</tr>
<tr>
<td>TestDissemination</td>
<td>135</td>
<td>9</td>
</tr>
<tr>
<td>Deluge-Blink</td>
<td>265</td>
<td>7</td>
</tr>
</tbody>
</table>

The above extension can potentially detect more system problems, especially those related to timing. However, it also introduces additional overhead to the runtime system. Therefore, we provide an option to turn on D2’s capability in detecting timing problems. In D2’s basic version, only function counts are collected. In D2’s extended version, additional I/O timing information is collected.

6 Evaluation

In this section, we present an evaluation of D2. Section 6.1 introduces the evaluation setup as well as the benchmarks we use. Section 6.2 evaluates D2’s overhead on the TelosB nodes including RAM overhead, program flash overhead, external flash overhead, and the CPU slowdown. Section 6.3 describes case studies in real sensor network applications.

6.1 Benchmarks

In order to evaluate the overhead of D2 on real sensor nodes, we investigate five typical benchmarks on TelosB nodes with 8MHz MSP430f1611 processor, 128KB program size, 10KB memory size, 1MB external flash size, and 250Kbps CC2420 radio.

All five benchmarks are in the TinyOS 2.1.1 distribution.

- TestNetwork: TestNetworkC uses the basic networking layers, CTP (collection) and Drip (dissemination).
- Oscilloscope: Oscilloscope is a simple data collection demo. It periodically samples the default sensor and broadcasts a message over the radio every 10 readings.
- TestDip: TestNetworkC exercises the basic dissemination protocol DIP.
- TestDissemination: TestDissemination exercises the basic dissemination protocol Drip.
- Deluge-Blink: Deluge-Blink is the simple Blink application with the Deluge reprogramming support.

Table 2 shows the number of principal components for the five benchmarks. We see a small number of dimensions can essentially capture large variance in the original data, indicating that the dimensions are highly correlated.

6.2 Overhead

This section evaluates the overhead of D2’s basic version on the TelosB nodes, including RAM overhead, program flash overhead, external flash overhead, and CPU slowdown.

Fig. 7: RAM consumption of TinyOS applications.

6.2.1 RAM

D2 requires a set of function counters to be allocated on RAM to track the number of each function’s executions. The RAM overhead is proportional to the number of functions to be tracked. We currently use 4 bytes for each counter. Besides, we need 4 additional bytes to track the total count of all function executions.

Table 3 shows the RAM overhead for five benchmark applications based on TinyOS 2.1.1. We can see that the RAM overhead varies from 544 bytes to 1064 bytes.

To see whether this RAM overhead is acceptable, we measure the RAM consumptions of the five benchmarks in Figure 7 (the statistics for GreenOrbs are similar to the TestNetwork application). The TinyOS applications consume 391–4386 bytes while the stacks\(^1\) consume 464–674 bytes. Since TinyOS employs static memory allocation (i.e., the remaining memory space will not be used by the OS and the applications), D2’s memory consumptions can well fit into the remaining memory space.

6.2.2 Program flash

D2 increases program flash size in two ways. First, the D2 module which performs binary instrumentation takes about 6.8kB program memory. Second, after instrumentation, the original program increases because of the trampoline overhead for each function.

The first overhead is a constant for all benchmarks. For all the benchmarks investigated in this paper, we can fit the D2 module on the program flash. When the original application code size is large, we can perform further optimizations by dynamic loading, i.e., storing the D2 module on the external flash and loading onto the program memory by

\(1\) We use the stack-estimator tool available at http://selab.csuohio.edu/stack-estimator.
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2542815, IEEE Transactions on Parallel and Distributed Systems

Fig. 8: Program flash overhead

Table 4: Increase of program size when function inlining is prohibited

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Default</th>
<th>non-inlining</th>
<th>inc. rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestNetwork</td>
<td>33.102</td>
<td>35.576</td>
<td>7.47%</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>17.450</td>
<td>18.860</td>
<td>8.08%</td>
</tr>
<tr>
<td>TestDip</td>
<td>17.154</td>
<td>18.582</td>
<td>8.32%</td>
</tr>
<tr>
<td>TestDissemination</td>
<td>14.826</td>
<td>16.278</td>
<td>9.79%</td>
</tr>
<tr>
<td>Deluge-Blink</td>
<td>32.866</td>
<td>35.208</td>
<td>7.13%</td>
</tr>
</tbody>
</table>

another smaller bootloader when needed. After D2 performs binary instrumentation, the node switches to the instrumented application code. In this way, the memory overhead of D2 module has no impact on the application.

The second overhead depends on the complexity of each benchmark. Figure 8 shows the original program size, the constant overhead of D2, and the trampoline overhead for five benchmarks. The trampoline overhead after instrumentation depends on the number of functions in the compiled code.

Function inlining is a common technique for optimizing program performance. However, it can hinder localizing to the correct function in D2. To strike a reasonable balance, we would require large functions non-inlined so that bugs can be traced to the corresponding function. Table 3 shows the increase of program size when we prohibit inlining of functions exceeding 200 bytes. We can see that the increase is below 10% compared with the default case with aggressive function inlining.

### 6.2.3 External flash

Usually, D2 needs to store snapshots of function counts onto the external flash for later analysis. The snapshots can also be directly sent to the sink node for real-time analysis. The snapshot overhead depends on how frequently D2 takes snapshots.

Table 5 show the snapshot overhead for five benchmark applications for 30 minutes when the threshold $\phi$ for the total function counter is set at 5000. We also compare the overhead of D2’s counter-based profiling with LIS’s event-based profiling [20], [21]. We can see that D2 results in 39–220 reduction.

Finally, it is important to examine the energy overhead on the external flash. According to our previous study,

- the current when reading from external flash is $I_{\text{read}} = 5\text{mA}$.
- the current when writing to external flash is $I_{\text{write}} = 12\text{mA}$.
- the time for reading one byte from the external flash is $T_{\text{read}} = 0.045\text{ms}$.
- the time for writing one byte to the external flash is $T_{\text{write}} = 0.059\text{ms}$.

Considering the per-byte energy consumption of logging on the external flash as well as reading from the flash for later collection, we can get: $W_{\text{byte}} = I_{\text{read}}T_{\text{read}} + I_{\text{write}}T_{\text{write}}$. Table 6 shows the energy consumption for the five benchmarks lasting for 30 minutes (with $\phi = 5000$). We also show the relative increase compared with the original application. We can see that the relative increase is small.

#### 6.2.4 CPU utilization

D2 slightly degrades program’s execution because of the overhead in the trampolines. Table 7 compares the CPU utilizations of the original program (without D2) and the instrumented program (with D2). We further differentiate two cases: 1) D2(ai), D2’s default case with aggressive inlining, and 2) D2(pi), prohibiting the inlining of functions exceeding 200 bytes. We can see that the increase of CPU utilization is small. This will not affect the network performance as most sensor network applications are not CPU-intensive.

Nevertheless, D2 may still introduce Heisenbugs due to binary instrumentation. We try to minimize the impact by minimizing D2’s run-time overhead. The degradation can be eliminated if additional hardware is employed. For example, if the AVEKSHA hardware [12] is employed, the extra CPU attaching to the JTAG will be interrupted at the start of each function and thus can execute the trampoline in parallel to the main CPU, eliminating the execution overhead of trampolines.

#### 6.2.5 Comparison with other approaches

Table 5: External flash overhead for 30 minutes (bytes) with $\phi=5000$.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>$\phi=5000$</th>
<th>$I_{\text{read}}T_{\text{read}} + I_{\text{write}}T_{\text{write}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestNetwork</td>
<td>15934</td>
<td>22.43</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>40688</td>
<td>55.55</td>
</tr>
<tr>
<td>TestDip</td>
<td>19536</td>
<td>81.15</td>
</tr>
<tr>
<td>TestDissemination</td>
<td>13230</td>
<td>77.7</td>
</tr>
<tr>
<td>Deluge-Blink</td>
<td>14310</td>
<td>39.6</td>
</tr>
</tbody>
</table>

Table 6: Additional energy consumption due to storing snapshots on external flash (mAs)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Without D2</th>
<th>Additional cost</th>
<th>inc. rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestNetwork</td>
<td>981.00</td>
<td>15.74</td>
<td>1.60%</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>2185.67</td>
<td>67.86</td>
<td>3.10%</td>
</tr>
<tr>
<td>TestDip</td>
<td>1118.34</td>
<td>19.3</td>
<td>1.73%</td>
</tr>
<tr>
<td>TestDissemination</td>
<td>627.84</td>
<td>13.07</td>
<td>2.08%</td>
</tr>
<tr>
<td>Deluge-Blink</td>
<td>965.30</td>
<td>14.13</td>
<td>1.46%</td>
</tr>
</tbody>
</table>

Table 7: Comparison of CPU utilizations

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Without D2</th>
<th>With D2(ai)</th>
<th>With D2(pi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestNetwork</td>
<td>2.16%</td>
<td>2.50%</td>
<td>2.78%</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>4.82%</td>
<td>5.57%</td>
<td>6.21%</td>
</tr>
<tr>
<td>TestDip</td>
<td>2.47%</td>
<td>2.85%</td>
<td>3.18%</td>
</tr>
<tr>
<td>TestDissemination</td>
<td>1.40%</td>
<td>1.60%</td>
<td>1.79%</td>
</tr>
<tr>
<td>Deluge-Blink</td>
<td>2.13%</td>
<td>2.46%</td>
<td>2.73%</td>
</tr>
</tbody>
</table>
We compare the overhead of D2 with LIS [20]—an event-based profiling method using the Oscilloscope benchmark. Table 8 shows the results. (1) CPU utilization. D2 increases the CPU utilization by 15% compared with the uninstrumented program. According to [11], LIS increases by CPU utilization by 21%. (2) RAM overhead. D2 consumes more memory than LIS due to its allocation of function counters. Note that we can further optimize D2’s memory consumption by allocating 2 bytes (instead of 4 bytes) for each function counter, as long as the function counter does not overfl ow during the maximum possible time window. (3) Size of diagnostic data (external flash overhead). D2 consumes significantly less overhead than LIS: LIS’s trace size is 56 times larger than the snapshot overhead of D2.

6.3 Case studies

Our goal in these studies is to demonstrate that D2 can be applied in existing sensor network applications and can simplify the complex process of detecting and diagnosing sensor network problems.

6.3.1 Case 1: flash broken

Application scenario. We test our GreenOrbs application program in an indoor testbed consisting of 50 TelosB nodes for 30 minutes. The GreenOrbs application basically reads the sensor data and delivers the data to the sink with a period of 1 minute. Each packet transmission and reception events are recorded on the external flash for later analysis.

Symptom. We find a symptom that two nodes did not record any events in their external flash, until we finish the experiment and attempt to retrieve the recorded events.

D2 setup and findings. In the next experiment, we turned on the D2 functionality for this application. The D2 profiles are collected through serial ports. We use 10 snapshots from each node to apply D2’s analysis approach.

D2 indeed detects that two nodes exhibit abnormal patterns. D2 can not only detect the problem in the network but also simplify the process of further diagnosis. Figure 9 depicts the diagnosis report generated by D2. We can see that four functions are frequently executed in abnormal nodes. Function runTask() invokes releaseAndRequest() which again invokes release() and request().

From the prefixes of these functions, we can easily guess that the STM25P component might be in error: the STM25P component repeatedly makes requests to acquire the SPI resource. Considering that these two nodes did not record any events in the external flash, we highly suspect that their external flash is broken and the buggy code does not take this condition into account.

Looking into the two files related to STM25P (i.e., Stm25pSpiPnc and Stm25pSectorPnc), we confirm our guess by finding that the powerUp() function would always return SUCCESS regardless of the status of the hardware (i.e., signature). The STM25P datasheet [38] indicates that the powering up succeeds only when the signature equals to 0x13.

Code fix. We fix the bug by checking the signature in the powerUp() function. If it does not succeed, we simply prevent the code from repeatedly acquiring the SPI resource.

Discussion. We note that this is a new bug which has not been previously reported as far as we know. We hope the bug will be fixed in the next release of TinyOS.

It is true that programmers can be alerted by examining the return value of write/read. However, this particular programmer had ignored the return value of a critical function in this case. D2 can detect the problem and narrow down the problem with a relatively small overhead in an automatic manner.

6.3.2 Case 2: TYMO routing protocol

Application scenario. We use 50 TelosB nodes in an indoor testbed. The application uses the TYMO routing protocol. TYMO is a TinyOS-based implementation of the DYMO routing protocol [39]—a successor of the AODV routing protocol [40]. In [23], authors reported a bug in the recent TYMO routing protocol in the TinyOS distribution. We apply D2 to a similar case to see whether D2 can also localize the bug. Sink node (node 50) is used to collect data. All other nodes (nodes 0, ... 48) are reporting packets to the sink via one hop wireless. The experiment lasts for 30 minutes.

Symptom. The sink node misses all packets from node 0.

D2 setup and findings. We turned on the D2 functionality and collected all the profiles for later analysis.

D2 generates the diagnosis report as shown in Figure 10. We can see that at node 0 the function getPayload() is called frequently while function getRoute() is called infrequently (compared with the normal behavior). Both functions are called in the selectRoute() function.

We look into the source code and find that the execution flow at node 0 diverges from the normal execution flow in...
Fig. 10: Diagnosis report for case 2. The size of a node represents the statistical significance. The upward arrow inside the node indicates a statistical increase. The arrow connecting two nodes represents a call/post relationship. The function selectRoute function (top) does not exhibits statistical difference in this case.

which a node tries to get a forwarding route and transmit the packet. Node 0 simply finds that the packet is for itself (the function isForMe(msg) returns true), causing the abnormal function executions in Fig. 10.

After more careful code reviews, we find that the real root cause is that the destination field is set too late: the destination field of msg is not set before calling selectRoute(). The function selectRoute() invokes isForMe() before msg is properly initialized. If the current node is 0, the function isForMe() will return true as the uninitialized fields in msg happen to be 0. So node 0 will not transmit any data. We can see that D2 can also localize the bug for this case.

**Code fix.** We fix the code by setting the destination field before calling selectRoute().

### 6.3.3 Case 3: CTP queue overflow

**Application scenario.** We deploy 50 TelosB nodes in the woodland of Zhejiang Forestry University. The nodes run the actual GreenOrbs application for one week.

**Symptom.** We have indeed detected a number of problems. In particular, nodes near the sink are more likely to experience heavier packet losses.

**D2 setup and findings.** We turned on the D2 functionality and collected all the profiles for later analysis. After distinguishing the anomaly points, we apply D2’s diagnosis method.

Figure 11 depicts the diagnosis report generated by D2. In this case, D2 finds that each individual function count does not exhibit much statistical difference. However, some ratios between functions exhibit large statistical difference. For example, the ratio between receive() and send() decreases in a few snapshots. D2 will automatically detect anomalies in function ratios when it finds each individual function count is normal.

This finding clearly indicates that the corresponding node during that time window experiences transient overflow so that the number of receive() and send() diverges. Looking into the code, we indeed find that the default CTP implementation does not turn on the congestion control mechanism.

**Code fix.** We implement a simple congestion control mechanism as follows. First, we set the ECN bit to notify neighbors when the current queue size exceeds the maximum allowable size. Second, after receiving packets with the ECN bit turned on, the current node will avoid selecting the congested node. More advanced congestion control mechanisms can be adopted. However, we find that this simple mechanism addresses this problem fairly well in practice.

**Discussion.** The PCA approach used by D2 can automatically find correlations among multiple dimensions. D2 can detect anomalies violating the correlations. For problem diagnosis, D2 will first check the function counts to pinpoint abnormal functions. If the function counts are normal, it will proceed to check the ratios of function counts to pinpoint which ratios are abnormal.

This case study gives us the following implications. First, some problems can be caused by the joint effects of code imperfections as well as the topology and traffic in real systems. Hence, a method that can be applied in a realworld deployed system is important to capture such problems. Second, sensor network exhibits variable workload. Simple analysis on each individual dimension may not be able to reveal some problems. We find PCA is suitable because it can capture the essence of the correlations between multiple dimensions.

### 6.3.4 Case 4: FTSP+CTP in TinyOS 2.1.0

**Application scenario.** We use 24 TelosB nodes in our indoor testbed for one hour. The application uses the CTP protocol to collect sensor data at a rate of 5 minutes. It also uses the FTSP protocol for global time synchronization.

**Symptom.** After 10 minutes, several nodes stop delivering data to the sink node.
D2 setup and findings. We repeat the experiments, and turn on the D2 functionality and collect all the profiles for later analysis. D2 generates the diagnosis report as shown in Figure 12. We can see that at node 6 three function counts decrease rapidly. All these functions are related to the CTP sending logic. This implies that node 6 stops invoking sendMessage() due to some reasons in the radio stack.

We look into the source code and find that the most likely reason is due to the busy flag sendBusy in the CTP components. Normally, sendBusy will be reset to FALSE in the sendDone() event. However, when CTP works together with FTSP, it is possible that the invocation to the MAC-layer send fails after sendBusy is set to TRUE. This, in turn, causes the sendDone() event never happen. Hence the CTP protocol hangs. It is correct when CTP is the sole protocol in the application because the invocation to the MAC-layer send always succeeds. However, multiple protocols with different purposes may co-exist in a sensor to fulfill different tasks. Hence, uncoordinated resource contention among different protocols may be transiently triggered and the system eventually fails [22]. This bug has been extensively discussed in the TinyOS mailing list.

Code fix. We fix the code by examining the return value of MAC-layer send(), and post the send() task in a retransmission timer until it succeeds.

6.3.5 Case 5: CTP in high-data-rate applications

Application scenario. We use 50 TelosB nodes in our indoor testbed for one hour. The application uses the CTP protocol to collect sensor data. We change the data transmission period from 10s to 1s.

Symptom. We find that at data transmission period of 10s, the sink can collect the sensor data. However, the sink receives no data when the transmission period is changed to 1s.

D2 setup and findings. We repeat the experiment, and turn on the D2 functionality. In particular, we use D2’s extended version to also collect the timing information of TinyOS split-phase I/Os. We suspect that there are problems in the radio transmission components. From Figure 13, we find that the I/O waiting times of two operations significantly deviate from the normal. One is the send() to sendDone() time interval in the CtpForwardingEngine component. The other is the ACK waiting time which is the time interval from the packet is actually transmitted (an SFD interrupt will be generated) to the corresponding ACK is received (another SFD interrupt corresponding to the ACK will be generated).

Figure 13 shows that the I/O waiting times of these two I/O operations with varying transmission periods. We can see that at 10s transmission period, the two I/O waiting times are kept small. However, both I/O waiting times significantly increase when the transmission period decreases (i.e., at a higher data rate). The large increase of ACK time (from 3ms to 20ms) clearly suggests an unexpected scenario from the perspective of developers: the TinyOS link layer will retransmit the packet after an ACK timeout of 1/128 ≈ 7.8ms. If the ACK is returned at a later time, the link layer will repeatedly retransmit the packet, making the CTP protocol to hang eventually. The increase of ACK time is due to the fact that TinyOS uses software ACK by default [41]. Software ACKs can be interrupted and delayed by other radio activities.

Code fix. We fix the problem by turning on the CC2420_HW_ACKNOWLEDGEMENTS flag to inform TinyOS to use the hardware ACKs instead of software ACKs.

6.3.6 Comparative study

We also perform a comparative study on both one-class SVM (used by Sentomist) and PCA (used by D2) for the four case studies described earlier (Section 6.3.1–6.3.4). We use two metrics for comparison: the overhead and the effectiveness. The overhead of SVM and PCA is the time taken by these approaches to detect an anomaly. The detection algorithm is run on a PC with a dual-core 2.3GHz CPU and 4GB RAM. Table 9 compares the execution times of SVM and PCA on the PCs. We can see that PCA runs slightly faster than SVM.

The end-to-end metric for evaluating the effectiveness of a diagnosis approach is the amount of developer time required to diagnose each bug. However, this metric highly depends on the expertise of the developer as well as his familiarity to different code components. We choose detection accuracy as an effectiveness metric since problem diagnosis will be more efficient if the anomalies are more accurately detected in the first place.

We manually inject bugs so that we can compare our result...
Table 9: Execution time (ms) of SVM and PCA.

<table>
<thead>
<tr>
<th>Cases</th>
<th>SVM</th>
<th>PCA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash broken</td>
<td>19.20</td>
<td>12.42</td>
</tr>
<tr>
<td>TYMO</td>
<td>19.92</td>
<td>17.66</td>
</tr>
<tr>
<td>CTP-overflow</td>
<td>23.01</td>
<td>20.78</td>
</tr>
<tr>
<td>CTP+FTSP</td>
<td>26.91</td>
<td>21.83</td>
</tr>
</tbody>
</table>

Table 10: Detection accuracy of SVM and PCA.

<table>
<thead>
<tr>
<th>Cases</th>
<th>SVM (FP/FN)</th>
<th>PCA (FP/FN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash broken</td>
<td>100% (0%/0%)</td>
<td>100% (0%/0%)</td>
</tr>
<tr>
<td>TYMO</td>
<td>90% (10%/10%)</td>
<td>100% (0%/0%)</td>
</tr>
<tr>
<td>CTP-overflow</td>
<td>90% (10%/10%)</td>
<td>90% (10%/10%)</td>
</tr>
<tr>
<td>CTP+FTSP</td>
<td>90% (10%/10%)</td>
<td>90% (10%/10%)</td>
</tr>
</tbody>
</table>

with the ground truth. In each case study, we perform one-class SVM and PCA on 250 data points with 10 known anomalies produced by the buggy program code. Table 10 shows detection accuracy which is defined as the ratio of the number of real anomalies among the 10 top ranked anomalies produced by two approaches. We also show the false positive ratio (FP) and false negative ratio (FN). We can see that PCA has a better accuracy than one-class SVM, especially for the third case.

We believe that the injected bugs in the case studies are representative since the injected bugs are the ones we have actually encountered during the development of GreenOrbs, a real-world sensor network system. It is worth noting that D2 can also detect non-reproducible anomalies as long as the program profile about the anomaly has been collected. Whether D2 can detect the anomaly depends on whether D2 has collected the program profile about the anomaly, instead of whether the anomaly is reproducible or non-reproducible.

7 Conclusion

In this paper, we propose a method towards automated post-deployment diagnosis in networked embedded systems by combining program profiling and symptom mining. As opposed to previous instrumentation methods which either incur a large overhead or demand special hardware, we employ binary instrumentation to perform lightweight function count profiling. The statistics at the function level provide us fine-grained information for detailed reasoning. Unlike previous methods which require application programmers’ efforts for providing specific network metrics, our method treats the program as a black box, thus is scalable for a wide range of applications. Based on the statistics, we employ PCA-based approach for automatically detecting network problems. Previous network-level diagnosis methods only detect network problems at the node level or link level, D2 is able to point programmers closer to the most likely causes by a novel approach combining statistical tests and program call graph analysis.

We implement our method based on TinyOS 2.1.1 and evaluate its effectiveness by case studies in the development of GreenOrbs. Results show that our method can aid programmers to diagnose problems quickly in real-world sensor network systems, and at the same time, incurs an acceptable overhead to the running system.

Acknowledgement

This work is supported by the National Science Foundation of China Grants No. 61472360.

References


[42] Luyao Luo received his B.Eng. degree from Central South University in 2015. He is currently a graduate student at Zhejiang University, China. His research interests include networked embedded systems and system diagnosis.

[43] Chun Chen received his Bachelor of Mathematics degree from Xiamen University, China, in 1981, and his M.S. and Ph.D. degrees in Computer Science from Zhejiang University, China, in 1995 and 2000, respectively. He is a professor in College of Computer Science, and the Director of Institute of Computer Software at Zhejiang University. His research interests include embedded system, image processing, computer vision, and CAD/CAM.

[44] Jiajun Bu received the B.S. and Ph.D. degrees in Computer Science from Zhejiang University, China, in 1995 and 2000, respectively. He is a professor in College of Computer Science and the deputy dean of School of Software Technology at Zhejiang University. His research interests include embedded system, mobile multimedia, and data mining. He is a member of the IEEE and the ACM.

[45] Xue Liu is an Associate Professor and William Dawson Scholar in the School of Computer Science, McGill University, Montreal, QC, Canada. He received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 2006. He received his B.S. degree in Mathematics and M.S. degree in Automatic Control both from Tsinghua University, China. His research interests are in computer networks and communications, smart grid, realtime and embedded systems, cyber-physical systems, data centers, and software reliability.

[46] Yunhao Liu received the B.S. degree in automation from Tsinghua University, Beijing, China, in 1995, and the M.S. and Ph.D. degrees in computer science and engineering from Michigan State University, East Lansing, MI, USA, in 2003 and 2004, respectively. He is a Professor, Dean of the School of Software, and a member of the Tsinghua National Lab for Information Science and Technology, Tsinghua University. Prof. Liu is serving as the Chair of ACM China Council.