D2: Anomaly Detection and Diagnosis in Networked Embedded Systems by Program Profiling and Symptom Mining

Wei Dong†, Chun Chen†, Jiajun Bu†, Xue Liu‡ and Yunhao Liu§
†*Zhejiang Provincial Key Laboratory of Service Robot, College of Computer Science, Zhejiang University, China*
‡*School of Computer Science, McGill University, Canada*
§*MOE Key Lab for Information System Security, School of Software, TNLIST, Tsinghua University, China*
Emails: {dongw, chenc, bjj}@zju.edu.cn, xueliu@cs.mcgill.ca, yunhao@greenorbs.com

***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. 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 anomaly detection and diagnosis method 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 to 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 is effective for detecting and diagnosing problems in real-world sensor network systems, and at the same time, incurs an acceptable overhead.**

***Keywords*-networked embedded systems; sensor networks; diagnosis; program profiling; symptom mining**

I. 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 [2], an adaptive low duty cycle MAC protocol, (2) MintRoute [3], a multihop routing protocol, (3) Deluge [4], 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 2009 [5]. The initial software components include (1) CTP [6], a multihop routing protocol, (2) FTSP [7], 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 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 [8], [9], [10], [11], interactive debugging [12], [13], tracing and logging [14], [15], [16], network-level diagnosis [17], [18], [19], [20], etc. 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 new anomaly detection and diagnosis method by combining program profiling and symptom mining. Today's embedded sensor software has a low visibility in exposing detailed execution behaviors. 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 (Principal Component Analysis) 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 real-world sensor network applications. Results show that our method incurs an acceptable overhead and is powerful in detecting and diagnosing real-world problems.

The contributions of this work are summarized as follows.

(1) We propose a novel method combining program profiling and symptom mining for detecting and diagnosing anomalies in networked embedded systems.
(2) We propose a novel approach combining statistical tests and program call graph analysis to point programmers closer to the most likely causes.
(3) 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 II describes the related work. Section III presents the design principles. Section IV shows the evaluation results, and finally, Section V concludes this paper and gives directions of future work.

II. Related Work

There are numerous research works related to diagnosis. We classify existing works into five main categories: pre-deployment tools, debugging tools, logging and tracing, network-level diagnosis, and diagnosis in other distributed systems.

Pre-deployment tools. T-Check [8] 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 [9] uses symbolic analysis to generate test cases for sensor network code. Sentomist [21] uses the number of executed instructions during interrupt handling intervals to find transient bugs. T-Morph (TinyOS application tomography) [22] 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. Both Sentomist and T-Morph are based on Avrora [23]—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. These tools are mainly used before deployment and they cannot utilize the invaluable runtime information after deployment.

Debugging tools. Clairvoyant [12] 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 [24] 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. StackGuard [25] is a more generic tool for detecting stack corruption caused by buffer overruns. Declarative Tracepoint [13] 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 [26] 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 non-volatile flash. DustMiner [27] identifies bugs in sensor network software by checking discriminative log patterns. DustMiner requires detailed logging which may not be available in many existing sensor network software components. Even if available, detailed logging incurs relatively large overhead. Sundaram et al. propose an efficient intra-procedural and inter-procedural control-flow tracing algorithm that generates the traces of all interleaving concurrent events [14]. AVEKSHA [15] is a hardware-software approach for tracing events in a non-intrusive manner. There tools expose detailed control-flow information at runtime. However, they either incur relatively large overhead or demand special hardware. Moreover, the identification of faulty behaviors still heavily depends on manual efforts.

Network-level diagnosis. Sympathy [17] collects multiple network metrics and uses a decision tree to localize the failures. PAD [18] uses lightweight network monitoring and Bayesian network based analysis to infer network failures and their causes. Agnostic Diagnosis [28] also collects multiple network metrics and uses anomaly detection on the correlation graph to discover silent failures. TinyD2 [19] uses the concept of self-diagnosis in which each sensor can join the fault decision process. TinyD2 [19] plants a finite state machine into each sensor node, enabling them to accordingly change the diagnosis state. LD2 [29] 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

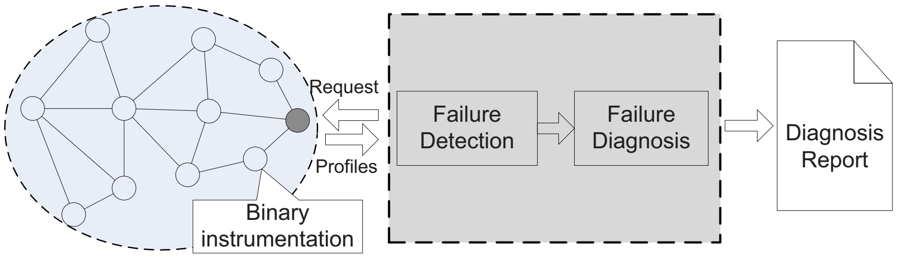


Figure 1: D2 overview

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.

Diagnosis in other distributed systems. Diagnosis in the Internet or distributed systems is also related to this work. Sherlock [30] presents a multi-level inference model that achieves high performance. Giza [31] addresses the performance diagnosis problem in a large IPTV network and NetMedic [32] enables detailed diagnosis in enterprise networks. Xu et al. mine console logs in data center servers to find performance anomalies [33]. Distalyzer [34] uses machine learning techniques to compare system behaviors extracted from the logs and automatically infer the strongest associations between system components and performance. Diagnosing based on statistical inference and machine learning techniques in the Internet/distributed systems give us important reference. However, they require large amount of information which is not affordable by resource-constrained sensor networks.

III. Design

In this section, we present the design of D2. Section III-A presents an overview of D2. Section III-B describes how D2 instruments code to obtain run-time information. Section III-C describes how D2 detects a problem based on the above information. Section III-D describes how D2 performs problem diagnosis into the function level.

A. Architecture Overview of D2

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 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.

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 issues a profiling request. Therefore, like Clairvoyant [12], D2 can be left on the node *after deployment*, to be used only when needed.
- D2 uses *lightweight* function count profiling. Counter-based profiling incurs much less overhead than event-based profiling.
- Despite lightweight, D2 provides function-level information that can *effectively* drill down the problem inside the node.
- 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, it can also be applied to other OSes since binary instrumentation within D2 operates at the binary level.

B. 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 [35].

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.

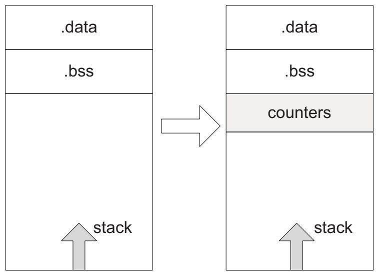


Figure 2: RAM layout on telosB nodes

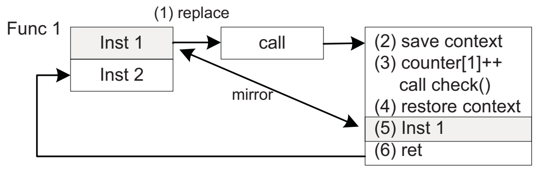


Figure 3: The trampoline technique

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 [36], 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 considering 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. A trampoline is a code fragment that is inserted into the target binary, such that the target execution is rerouted to the new code fragment, before it returns for the execution of the original code [35].

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 external flash.
(4) The context is restored by poping 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 [37] 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.

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 [38] or DIP [39].

C. Problem Detection

The D2 module on the PC is responsible for detecting the problems. D2 retrieves the snapshots by either wireless communications or serial connections.

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. Based on the function count snapshots, D2 constructs features for problem detection. Function counts encode the CPU activities. 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 [6] 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.

To encode this correlation, we construct function count vectors **f**. Each function count vector represents a group of functions in a time window, while each dimension of the vector corresponds to a distinct function, and the value of the dimension is how many times this function is executed in the time window.

We combine all n-dimensional **f**'s from m time windows from N network nodes to construct the $m \times n$ function count matrix **F**. Note that n is the number of functions in the program and can be obtained in the binary instrumentation process.

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 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). The runtime overhead of PCA is linear with the number of feature vectors and thus can scale to large data.

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 $\mathbf{S}_n$. The axis S_n captures the strong correlation between the two dimensions. Therefore, data points far from $\mathbf{S}_n$, 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 $\mathbf{S}_n$, 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.

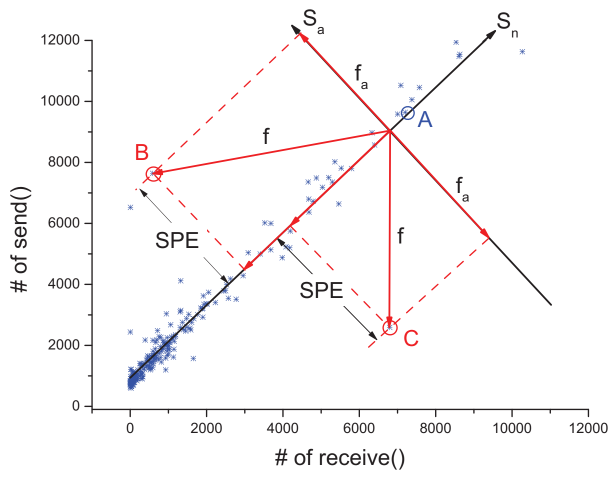


Figure 4: PCA anomaly detection. S_n indicates the normal subspace and S_a indicates the abnormal subspace. Points B and C are identified as anomalies.

Table 1: 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.

Data sets	n	k
TestDissemination	93	1
Block	58	1
RadioCountToLeds	82	4

Table 1 shows the number of principal components in three sensor network applications. We see a small number of dimensions can essentially capture large variance in the original data, indicating that the dimensions are highly correlated.

Like [33], we use the distance from a data point to the normal subspace $\mathbf{S}_n$ to determine whether **f** is an anomaly. The squared prediction error $\mathbf{SPE} = ||\mathbf{f}_a||$ where $\mathbf{f}_a = (\mathbf{I} - \mathbf{PP}^T)\mathbf{f}$ is the projection of **f** onto the abnormal subspace S_a, and $\mathbf{P} = [\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_k]$ where $\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_k$ are the principal components.

We use a threshold of Q_α to detect whether a data point is abnormal:

$$\mathbf{SPE} = ||\mathbf{f}_a||^2 > Q_\alpha \quad (1)$$

where Q_α denotes the Q statistic (a well known test statistic for the SPE residual function [40]) at the $1-\alpha$ confidence level. The choice of the confidence parameter α for anomaly detection has been studied in previous work [41]. We choose $\alpha = 0.001$ as in previous work [33].

D. Problem Diagnosis

Now we have detected anomalies in time windows on specific nodes. We still lack detailed information why the anomaly occurs. We notice that in practical systems, unusual patterns in low dimensions often contribute to the formation of anomaly in a high dimension. For example, we have detected that during a time window w_1, node A is abnormal by examining its function count feature vectors with 200 dimensions. This symptom might be caused by only a few frequently executed functions.

There are circumstances that an anomaly occurs with each individual dimension being normal. The correlations among multiple dimensions, however, exhibit abnormal patterns. For example, we have detected that during a time window w_2, node B is abnormal by examining its function count feature vectors with 200 dimensions. The cause of this symptom is due to the decrease of the ratio between receive() and send().

In the above two cases, reducing the high dimension to the low dimension with the most suspicious functions will further narrow down the problem. Given an anomaly and a set of normal profiles, we would like to examine each individual dimension and the ratio between two dimensions to see whether they are significantly different from normal data.

We perform t-tests [42] to compare n function counts and n^2 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 [34].

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.

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.

IV. Evaluation

In this section, we present an evaluation of D2. Section IV-A evaluates D2's overhead in terms of RAM consumption, program flash consumption, external flash consumption, and CPU slowdown. Section IV-B describes case studies in real sensor network application.

A. Overhead

1) RAM overhead: 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 2 shows the RAM overhead for 5 benchmark applications based on TinyOS 2.1.1. We can see that the RAM overhead varies from 196 bytes to 632 bytes. Overall, the RAM overhead is acceptable since telosB has a total of 10KB RAM.

Table 2: RAM overhead (bytes)

Benchmarks	# of func	Overhead
Blink	48	196
RadioCountToLeds	70	284
TestDissemination	85	344
TestNetwork	157	632
Oscilloscope	101	408

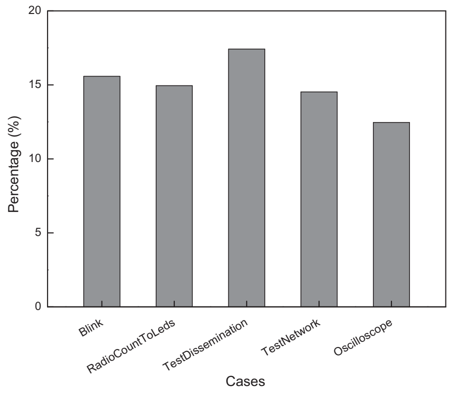


Figure 5: The instrumentation overhead relative to the program size

2) Program flash: D2 increases program flash size in two ways. First, the D2 module which performs binary instrumentation takes about 10KB program memory. Second, after instrumentation, the original program increases because of the trampoline overhead for each function.

The first overhead can further be optimized by storing the D2 module on the external flash and loading onto the program memory by 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.

Figure 5 shows the ratio of trampoline overhead and the original program size. We can see that the relative increase is about 10%. This overhead depends on the number of functions in the compiled code. Function inlining can reduce the overhead, but less information can be collected since D2 only counts at the granularity of function. If we demand finer-grained diagnosis, we can either simply disable function inlining or adopt more advanced control-flow profiling methods [14] at the cost of larger execution overhead.

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. Tables 3 shows the snapshot overhead for 5 benchmark applications for 30 minutes when the count of all function executions ϕ is set at 2000 and 5000. This

Table 3: External flash overhead for 30 minutes (bytes) with ϕ=2000 and ϕ=5000.

Benchmarks	ϕ=2000	ϕ=5000
Blink	3000	1140
RadioCountToLeds	4941	1757
TestDissemination	9272	3722
TestNetwork	16241	6280
Oscilloscope	90447	40400

Table 4: Comparison of CPU utilizations

Benchmarks	Without D2	With D2
Blink	1.38%	1.59%
RadioCountToLeds	1.22%	1.45%
TestDissemination	1.40%	1.60%
TestNetwork	2.16%	2.50%
Oscilloscope	4.82%	5.57%

overhead can further be reduced if trace compression [43] is adopted.

4) CPU slowdown: D2 slightly degrades program's execution because of the overhead in the trampolines. Table 4 compares the CPU utilizations of the original program and the instrumented program. 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.

On the other hand, D2's current implementation may 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 [15] 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.

B. 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.

We report two case studies. We have identified one unreported code bug in the latest TinyOS code (case 1). We also describe how D2 can detect networking problems (case 2).

1) Case 1: flash broken: During our indoor testing of the GreenOrbs application [5], we turned on local logging to examine the detailed application behaviors.

We test a total of 50 nodes with D2 monitoring the system performance (see Figure 6). We obtain the D2 profiles through serial ports. We use 10 snapshots from each node to apply D2's analysis approach.

We indeed find that two nodes exhibit abnormal patterns. Further problem diagnosis requires manual inspection to each abnormal point with nearly 280 function counts.

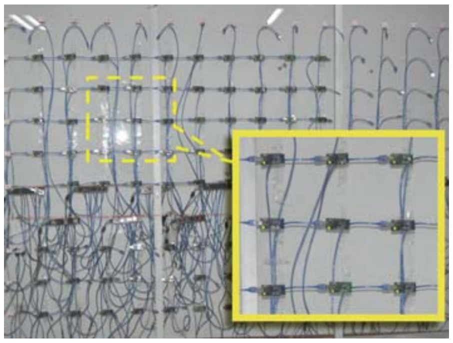
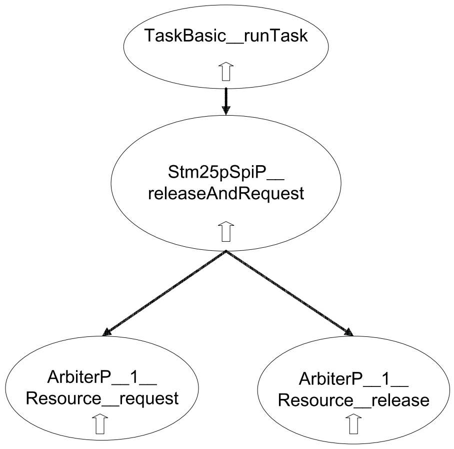


Figure 6: A testbed consisting of 50 telosB nodes for case 1


TaskBasic__runTask
Stm25pSpiP__ releaseAndRequest
ArbiterP__1__ Resource__request
ArbiterP__1__ Resource__release
Figure 7: Diagnosis report for case 1. 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.

D2 can not only detect the problem in the network but also simplify the process of further diagnosis. Figure 7 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().

Looking into the source code, we could guess the causes of the bug: when the code powers up the external flash, it does not check the status of the hardware. Therefore, if the external flash is broken, the code would repeatedly make requests to acquire the resource.

We indeed find the bug in the Spi.powerUp() function. The function always returns SUCCESS without checking the value of signature. We check the STM25P datasheet [44] and find that the powering up succeeds only when the signature equals to 0x13.

We have fixed this bug in our GreenOrbs application. This bug still exists in the latest TinyOS code and we plan to make notifications to the TinyOS development group.

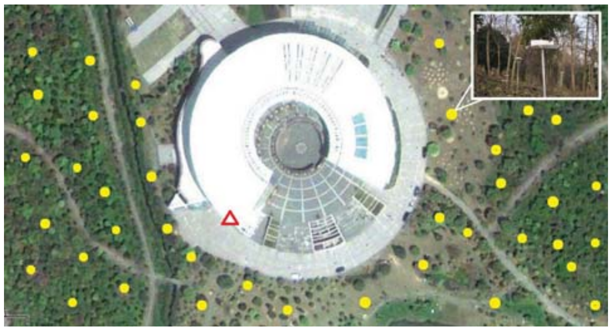


Figure 8: Deployment for case 2. The triangle denotes the sink node. The circles denote other nodes.

It is worth mentioning that although the bug is deterministic in this case, it is hard to detect using a single node. First, the proportion of broken nodes is fairly low (e.g., 2 out of 50). Second, even with a broken node, the problem cannot be easily localized.

This case study gives us the following implications. First, some errors are hard to detect unless a complete set of system metrics are incorporated into the system. For example, before this experiment, we only use the packet delivery ratio (PDR) from each node as a system metric and hence can not detect the problem. This problem is, however, important as it can greatly affect the energy efficiency of the system. Applying statistical methods can catch “bugs under the nose” without domain knowledge. PCA can detect the problem in this case because the abnormal symptom appears infrequent. Second, when we already have knowledge on the correctness of a profile (e.g., according to a system metric). We can directly classify the profiles into two classes, i.e., normal and abnormal. D2 is still helpful because it further drills down the problem to function level with call/post relationships.

2) Case 2: CTP queue overflow: We have applied the D2 method to a small deployment of GreenOrbs with 50 telosB nodes in the woodland of Zhejiang Forestry University (see Figure 8).

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

We have indeed detected a number of problems. In particular, nodes near the sink are more likely to experience problems. After distinguishing the anomaly points, we apply D2’s diagnosis method. Figure 9 depicts the diagnosis report generated by D2.

In this case, we find 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.

This symptom clearly indicates that the corresponding node during that time window experiences transient overflow so that the number of receive() and send() diverges.

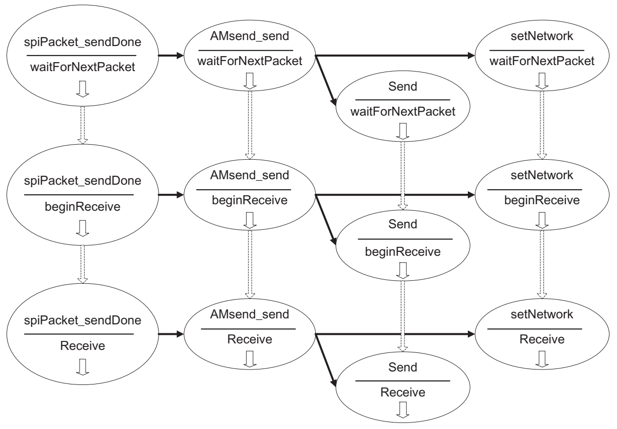


Figure 9: Diagnosis report for case 2. The size of a node represents the statistical significance of the ratio. The downward arrow indicates a statistical decrease. A solid arrow connecting two nodes indicates that there is a call/post relationship between the functions in the numerator while a blank arrow connecting two nodes indicates that there is a call/post relationship between functions in the denominator.

Looking into the code, we indeed find that the default CTP implementation does not turn on the congestion control mechanism. The TinyOS code at that time does not have a complete implementation of this mechanism. We implement this mechanism as follows. First, we set the ECN bit to notify neighbors when the current queue size exceeds the maximum 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.

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 real-world 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.

V. Conclusion

In this paper, we propose a new anomaly detection and diagnosis method 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 is effective in detecting and diagnosing problems in real-world sensor networks, and at the same time, incurs an acceptable overhead.

D2 has limitations. First, D2's function count profiling approach, albeit lightweight, loses 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 [37], at the cost of large profiling overhead. Second, D2 cannot detect concurrency problems due to improper interleavings of functions. Detecting such bugs requires detailed function-level logging [27] to track the order of function executions. Third, in some cases, D2 cannot detect the problem solely based on the statistical difference. For example, in TinyOS 2.1.0, the Trickle timer bug causes all nodes to frequently execute the fired() event after 10 minutes. D2 cannot differentiate between normal and abnormal patterns in this case. It would be useful to incorporate a performance metric like CPU utilization for problem detection. Fourth, D2 does not consider timing information which we would like to take into account in our future work.

Acknowledgment

This work is supported by the National Science Foundation of China Grants No. 61202402 and 61070155, the National Key Technology R&D Program (2012BAI34B01), the Fundamental Research Funds for the Central Universities (2012QNA5007), the Research Fund for the Doctoral Program of Higher Education of China, and the NSFC Distinguished Young Scholars Program under Grant No. 61125202.

References

[1] K. Langendoen, A. Baggio, and O. Visser, "Murphy Loves Potatoes: Experiences from a Pilot Sensor Network Deployment in Precision Agriculture," in *Proc. of the International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, 2006.

[2] T. van Dam and K. Langendoen, "An Adaptive Energy-Efficient MAC Protocol for Wireless Sensor Networks," in *Proc. of ACM SenSys*, 2003.

[3] A. Woo, T. Tong, and D. Culler, "Taming the Underlying Challenges of Reliable Multihop Routing in Sensor Networks," in *Proc. of ACM SenSys*, 2003.

[4] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proc. of ACM SenSys*, 2004.

[5] Y. Liu, Y. He, M. Li, J. Wang, K. Liu, L. Mo, W. Dong, Z. Yang, M. Xi, and J. Zhao, "Does Wireless Sensor Network Scale? A Measurement Study on GreenOrbs," in *Proc. of IEEE INFOCOM*, 2011.

[6] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection Tree Protocol," in *Proc. of ACM SenSys*, 2009.

[7] M. Maróti, B. Kusy, G. Simon, and Á. Lédeczi, "The Flooding Time Synchronization Protocol," in *Proc. of ACM SenSys*, 2004.

[8] P. Li and J. Regehr, "T-Check: Bug Finding for Sensor Networks," in *Proc. of ACM/IEEE IPSN*, 2010.

[9] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, "KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment," in *Proc. of ACM/IEEE IPSN*, 2010.

[10] B. Bonakdarpour and S. S. Kulkarni, "Compositional verification of real-time fault-tolerant programs," in *Proc. of EMSOFT*, 2009.

[11] L. Mottola, T. Voigt, L. Baresi, and C. Ghezzi, "Anquiro: Enabling Efficient Static Verification of Sensor Network Software," in *Proc. of the 1st International Workshop on Software Engineering for Sensor Networks*, 2010.

[12] J. Yang, M. L. Soffa, L. Selava, and K. Whitehouse, "Clairvoyant: A comprehensive source-level debugger for wireless sensor networks," in *Proc. of ACM SenSys*, 2007.

[13] Q. Cao, T. Abdelzaher, J. Stankovic, and L. Luo, "Declarative Tracepoints: A Programmable and Application Independent Debugging System for Wireless Sensor Networks," in *Proc. of ACM SenSys*, 2008.

[14] V. Sundaram, P. Eugster, and X. Zhang, "Efficient Diagnostic Tracing for Wireless Sensor Networks," in *Proc. of ACM SenSys*, 2010.

[15] M. Tancreti, M. Hossain, S. Bagchi, and V. Raghunathan, "Aveksha: A Hardware-Software Approach for Non-intrusive Tracing and Profiling of Wireless Embedded Systems," in *Proc. of ACM SenSys*, 2011.

[16] R. Sauter, O. Saukh, O. Frietsch, and P. J. Marrón, "TinyLTS: Efficient Network-Wide Logging and Tracing System for TinyOS," in *Proc. of IEEE INFOCOM*, 2011.

[17] N. Ramanathan, K. Chang, L. Girod, R. Kapur, E. Kohler, and D. Estrin, "Sympathy for the sensor network debugger," in *Proc. of ACM SenSys*, 2005.

[18] Y. Liu, K. Liu, and M. Li, "Passive Diagnosis for Wireless Sensor Networks," *IEEE/ACM Transactions on Networking*, vol. 18, no. 4, pp. 1132–1144, 2010.

[19] K. Liu, Q. Ma, X. Zhao, and Y. Liu, "Self-Diagnosis for Large Scale Wireless Sensor Networks," in *Proc. of IEEE INFOCOM*, 2011.

[20] K. Römer and J. Ma, "PDA: Passive Distributed Assertions for Sensor Networks," in *Proc. of ACM/IEEE IPSN*, 2009.

[21] Y. Zhou, X. Chen, M. R. Lyu, and J. Liu, "Sentomist: Unveiling Transient Sensor Network Bugs via Symptom Mining," in *Proc. of IEEE ICDCS*, 2010.

[22] Y. Zhou, X. Chen, M. R. Lyu, and J. Liu, "T-Morph: Revealing Buggy Behaviors of TinyOS Applications via Rule Mining and Visualization," in *Proc. of SIGSOFT 2012/FSE 20*, 2012.

[23] B. L. Titzer and et al., "Avrora: Scalable Sensor Network Simulation With Precise Timing," in *Proc. of ACM/IEEE IPSN*, 2005.

[24] V. Krunic, E. Trumpler, and R. Han, "NodeMD: Diagnosing node-level faults in remote wireless sensor," in *Proc. of ACM MobiSys*, 2007.

[25] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. of USENIX Security Symposium*, 1998.

[26] L. Luo, T. He, G. Zhou, L. Gu, T. F. Abdelzaher, and J. A. Stankovic, "Achieving repeatability of asynchronous events in wireless sensor networks with EnviroLog," in *Proc. of IEEE INFOCOM*, 2006.

[27] M. Khan, H. K. Le, H. Ahmadi, and T. Abdelzaher, "DustMiner: Troubleshooting Interactive Complexity Bugs in Sensor Networks," in *Proc. of ACM SenSys*, 2008.

[28] X. Miao, K. Liu, Y. He, Y. Liu, and D. Papadias, "Agnostic Diagnosis: Discovering Silent Failures in Wireless Sensor Networks," in *Proc. of IEEE INFOCOM*, 2011.

[29] Q. Ma, K. Liu, X. Miao, and Y. Liu, "Sherlock is Around: Detecting Network Failures with Local Evidence Fusion," in *Proc. of IEEE INFOCOM*, 2012.

[30] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies," in *Proc. of ACM SIGCOMM*, 2007.

[31] A. Mahimkar, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and Q. Zhao, "Towards Automated Performance Diagnosis in a Large IPTV Network," in *Proc. of ACM SIGCOMM*, 2009.

[32] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl, "Detailed Diagnosis in Enterprise Networks," in *Proc. of ACM SIGCOMM*, 2009.

[33] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting Large-Scale System Problems by Mining Console Logs," in *Proc. of ACM SOSP*, 2009.

[34] K. Nagaraj, C. Killian, and J. Neville, "Structured Comparative Analysis of System Logs to Diagnose Performance Problems," in *Proc. of USENIX NSDI*, 2012.

[35] M. Ekman and H. Thane, "Dynamic Patching of Embedded Software," in *Proc. of IEEE RTAS*, 2007.

[36] TIS Committee, "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2," 1995.

[37] J. Ha, C. J. Rossbach, J. V. David, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel, "Improved Error Reporting for Software that Uses Black-Box Components," in *Proc. of ACM PLDI*, 2007.

[38] G. Tolle and D. Culler, "Design of an Application-Cooperative Management System for Wireless Sensor Networks," in *Proc. of EWSN*, 2005.

[39] K. Lin and P. Levis, "Data Discovery and Dissemination with DIP," in *Proceedings of ACM/IEEE IPSN*, 2008.

[40] J. E. Jsackson and G. S. Mudholkar, "Control procedures for residuals associated with principal component analysis," *Technometrics*, vol. 21, no. 3, pp. 341–349, 1979.

[41] M. C. A. Lakhina and C. Diot, "Diagnosing network-wide traffic anomalies," in *Proc. of ACM SIGCOMM*, 2004.

[42] S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.

[43] V. Sundaram, P. Eugster, and X. Zhang, "Prius: Generic Hybrid Trace Compression for Wireless Sensor Networks," in *Proc. of ACM SenSys*, 2012.

[44] M25P80 datasheet, STMicroelectronics, 2004.