SPI-SNOOPER: A Hardware-Software Approach for Transparent Network Monitoring in Wireless Sensor Networks

Mohammad Sajjad Hossain*
Google Inc.
Mountain View, CA, USA
sajjad@google.com

Woo Suk Lee
Purdue University
West Lafayette, IN, USA
lee992@purdue.edu

Vijay Raghunathan
Purdue University
West Lafayette, IN, USA
vr@purdue.edu

ABSTRACT

The lack of post-deployment visibility into system operation is one of the major challenges in ensuring reliable operation of remotely-deployed embedded systems such as wireless sensor nodes. Over the years, many software-based solutions (in the form of debugging tools and protocols) have been proposed for in-situ system monitoring. However, all of them share the trait that the monitoring functionality is implemented as software executing on the same embedded processor that the main application executes on. This is a poor design choice from a reliability perspective. This paper makes the case for a joint hardware-software solution to this problem and advocates the use of a dedicated reliability co-processor that is tasked with monitoring the operation of the embedded system. As an embodiment of this design principle, this paper presents SPI-SNOOPER, a co-processor augmented hardware platform specifically designed for network monitoring. SPI-SNOOPER is completely cross-compatible with the TELOS wireless sensor nodes from an operational standpoint and is based on a novel hardware architecture that enables transparent snooping of the communication bus between the main processor and the radio of the wireless embedded system. The accompanying software architecture provides a powerful tool for monitoring, logging, and even controlling all the communication that takes place between the main processor and the radio. We present a rigorous evaluation of our prototype and demonstrate its utility using a variety of usage scenarios.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Wireless communication, Distributed networks; C.2.3 [**Network Operations**]: Network monitoring

General Terms

Design, Experimentation, Reliability

Keywords

Wireless sensor networks, reliability, hardware-software co-design, co-processor, monitoring, snooping

*This was performed when the author was at Purdue University.

CODES+ISSS'12, October 7-12, 2012, Tampere, Finland.

1. INTRODUCTION

As networked embedded systems transition from lab-scale research prototypes to large-scale commercial deployments, providing reliable and dependable system operation becomes absolutely crucial to ensuring widespread adoption and commercial success. Due to the fact that sensor nodes operate in dynamic and unpredictable physical environments that cannot be recreated in a lab setting, it is now generally accepted that pre-deployment testing alone (using simulators, emulators, *etc*.) is not sufficient to guarantee reliability and that *in-situ monitoring of nodes after deployment is a must*. In accordance with this belief, a number of techniques have been proposed for post-deployment node-monitoring and control ([10, 13, 26, 27] are a few examples). However, all of them share the common trait that the node-monitoring functionality is implemented as software executing on the same embedded processor that the application executes on.

From a reliability perspective, this is a poor design choice due to several reasons. First, the monitoring software shares (and competes for) resources such as CPU cycles and memory space with the main application software, further depriving the application of these already-scarce resources. Second, the presence of this additional software can perturb the timing behavior of the application, possibly suppressing some subtle bugs, or causing a large slowdown in application execution. Third, such an architecture inherently has a single point of failure; *e.g.*, if the processor hangs or freezes (possibly due to a bug in the main application code), the monitoring software is rendered essentially useless.

We believe that a joint hardware-software approach is both required and ideal to address the problem of post-deployment node monitoring and control. Particularly, we advocate a rethinking of the hardware architecture of sensor nodes to include an additional (low-cost and low-power) component that we call the *reliability co-processor*, which is responsible for monitoring the operation of the sensor node. As we will show, logically and physically separating the monitoring functionality from both the application software and the main processor in this manner allows the monitoring to be conducted in a decoupled, non-intrusive, and transparent manner, enhancing reliability. This paper presents an embodiment of the above design principle in the form of SPI-SNOOPER, a co-processor augmented hardware architecture specifically designed for network monitoring. We select network monitoring because, as we will demonstrate, carefully monitoring the bi-directional communication activity in a sensor node can reveal a significant amount of information about its operation. However, our design (and more generally, any reliability co-processor augmented design) is certainly not limited to network monitoring alone and can be used for a variety of other scenarios as well. Specifically, this paper makes the following contributions:

- We present SPI-SNOOPER, the first wireless sensor node platform that integrates a reliability co-processor into its hardware architecture. SPI-SNOOPER provides many novel hardware features: (a) the reliability co-processor can passively monitor (*i.e.,* snoop) all the transactions on the Serial-Peripheral Interface (SPI) bus that connects the main processor and the radio on our sensor node. This enables the co-processor to have complete visibility into all the information transmitted and received by the node. The snooping is fully transparent in the sense that the main processor and radio are not aware that the bus is being monitored, (b) in addition to passively monitoring processor-radio communication, the co-processor also has the ability to disconnect the main processor from the SPI bus and take control of the bus. In addition to cutting off a processor that exhibits faulty behavior, this also allows the co-processor to independently transmit and receive packets to/from other nodes, (c) the reliability co-processor can also access other hardware components on the sensor node, (*e.g.*, read various sensors). This allows the co-processor to possibly (independently) validate the behavior of the main application software, and (d) the co-processor can reset/reboot the main processor if desired (*e.g.*, if it detects that the main processor has hung or is operating using corrupted state information).
- SPI-SNOOPER features a lightweight, yet powerful software architecture that allows it to exploit the unique hardware features in an accurate, reliable, and transparent manner.
- We design, implement, and evaluate several usage scenarios for the SPI-SNOOPER platform and demonstrate how it can be used to significantly enhance the level of post-deployment visibility and control that network designers have over remotely deployed wireless sensor nodes.

The remainder of this paper is organized as follows. Section 2 and 3 present the novel hardware and software architecture and implementation of SPI-SNOOPER respectively. Section 4 describes a rigorous evaluation that we performed on SPI-SNOOPER and the various usage scenarios that we implemented to demonstrate the utility and capability of the SPI-SNOOPER architecture. Section 5 describes related work and Section 6 concludes the paper with some discussion and avenues for future work.

2. SPI-SNOOPER HARDWARE DESIGN

As mentioned in Section 1, SPI-SNOOPER features a reliability co-processor augmented hardware architecture that enables it to monitor and control network communication in a manner that is transparent to the main-processor. This section describes the SPI-SNOOPER hardware architecture and its implementation in detail.

For compatibility reasons, we decided to base our hardware architecture around an existing wireless sensor node platform. Although there are numerous such platforms available [1], we picked the TELOS mote due to its widespread adoption and use in the wireless sensor network research community. Because, in the TELOS design, main processor and radio IC are placed separately and communicate with each other via SPI bus which is exposed as a trace on PCB, it is possible to physically tap into the bus without modifying the original hardware architecture of the TELOS mote. However, physical access to the SPI interface of an off-the-shelf TELOS mote seems impossible in practice due to the delicate and narrow traces on the compact form factor PCB. Therefore, we built our own PCB implementation of SPI-SNOOPER, which was based on the existing TELOS schematics [3], suitably enhanced with the co-processor and the additional logic required for monitoring and controlling the SPI bus that connects the main processor and the radio.

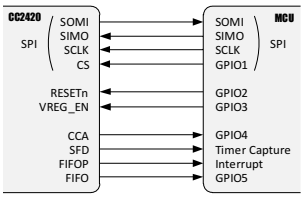


Figure 1: The interface between the CC2420 radio and the main processor on the TELOS mote. In all, ten lines are used.

2.1 The TELOS Mote

The TELOS [19] mote is a popular, open-source platform that features a Texas Instruments (TI) MSP430F1611 microcontroller as the main processor and an IEEE 802.15.4 compliant TI CC2420 radio transceiver. It also has on-board temperature, humidity, and light sensors and 1MB of external flash for data logging.

The hardware interface between the main processor and the radio on the TELOS is depicted in Figure 1. As shown in the figure, the main processor and the radio utilize a total of ten data and control lines for interfacing. They exchange data over an SPI bus using four pins (`SIMO`, `SOMI`, `SCLK`, and `CSn`). SPI is a full-duplex synchronous serial data link standard that supports communication between devices in a master/slave configuration. In the case of the TELOS mote, the processor acts as the master and initiates all bus transactions, while the CC2420 radio acts as the slave. Although the CC2420 radio cannot initiate bus transfers, it utilizes four other control lines (`SFD, CCA, FIFOP,` and `FIFO`) for reporting events that occurred in the radio. For example, `FIFO, FIFOP`, and `SFD` pins are used to notify the main processor of a received packet. In addition, the main processor can reset the radio and control the voltage regulator inside the radio via the `RESETn` and `VREG_EN` pins.

2.2 SPI-SNOOPER Hardware Architecture

As shown in Figure 2 and Figure 3, the SPI-SNOOPER hardware platform consists of four major components: main processor, reliability co-processor, radio, and crossover logic. The main processor is a TI MSP430F1611 that is connected to subsidiary components such as a data flash chip, serial ID chip, LEDs, switches, and sensors in an identical manner as the TELOS mote. For the radio, we utilize an off-the-shelf evaluation module (CC2420EMK from TI) in order to avoid the intricacies of sensitive radio-frequency circuit design. The crossover logic (described further in Section 2.5) is physically located between the main processor and the radio and splits the SPI bus connecting them into two segments. However, if the crossover logic is appropriately configured to simply act as a pass-through, the main processor gets connected to the radio module and becomes functionally identical to the TELOS platform. The dashed line in Figure 2 and Figure 3 represents the circuitry designed directly based on the TELOS mote's schematic [3] (except for a part of the crossover logic that also lies inside this area).

The rest of the SPI-SNOOPER board mainly houses the reliability co-processor and its related peripherals. The additional peripherals, including some LEDs, switches, USB, and JTAG, are mainly used for convenient programming, testing, and debugging of the reliability co-processor. The co-processor can also control the reset line of the main processor, which gives it the ability to reboot the main processor if desired. In addition, the co-processor can independently access the light sensors present on the board through wire traces branched from the sensors. The SPI-SNOOPER board use a nominal supply voltage of 3V and can be powered by either the MSP-FET430UIF [23], or two AA batteries, or the USB port similar to how the TELOS mote can be powered.

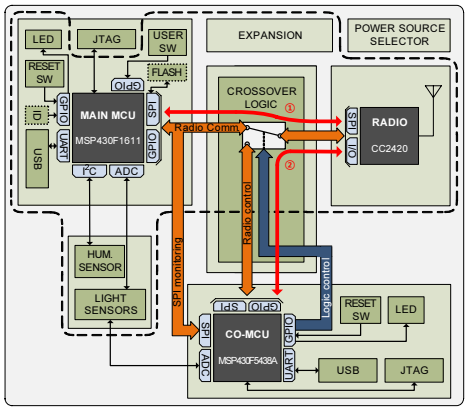


Figure 2: Block diagram of the SPI-SNOOPER platform.

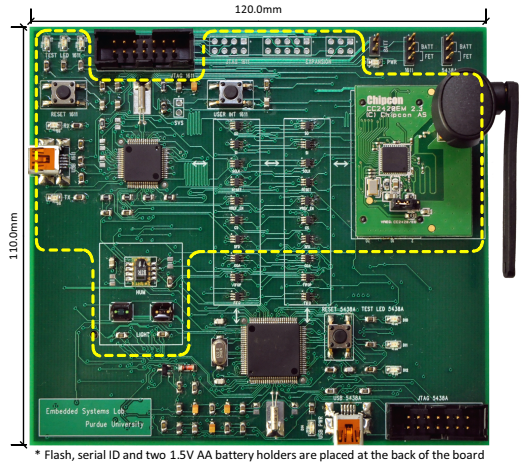


Figure 3: Image of an actual SPI-SNOOPER board. The dashed yellow border (roughly) encloses the components found on the TELOS mote.

2.3 Reliability co-processor: MSP430F5438A

The main considerations in selecting which microcontroller to use as the reliability co-processor are that it should be low-cost, low-power, provide sufficient computational resources, be easy to integrate into a larger design with minimal complexity, be easy to program, and have all the peripherals required to support the snooping of the SPI bus. After a survey of the several off-the-shelf MCUs, we selected the TI MSP430F5438A as the reliability co-processor for SPI-SNOOPER as it meets all the requirements above.

The MSP430F5438A is a 16-bit microcontroller that can run at a maximum frequency of 25MHz. It has three 16-bit timers, a 12-bit analog-to-digital (A/D) converter, up to four universal serial communication interfaces (USCI), and up to 87 general purpose IO pins [24]. The generous USCI and GPIO count are particularly crucial requirements, as we will see shortly. An added advantage of selecting this microcontroller is that it uses the same MSP430 instruction set architecture as the main processor on the TELOS, which eases programmer effort (*e.g.*, the same software toolchain can be used for both processors). It is also more power efficient than the main processor, *e.g.*, the MSP430F5438A consumes 350μA at 1MHz with 3V supply voltage in the active mode (AM), whereas the MSP430F1611 consumes 500μA under the same condition [24, 22]. The power consumption of SPI-SNOOPER will further be analyzed in section 4.1.5.

It should be noted that a microcontroller used as a reliability co-processor is not necessarily the MSP430F5438A. Although the MSP430F5438A was chosen due to its adequate performance, any microcontrollers which are able to run at over 20MHz would be a proper choice because the maximum SPI clock frequency of the CC2420 radio is 10MHz.

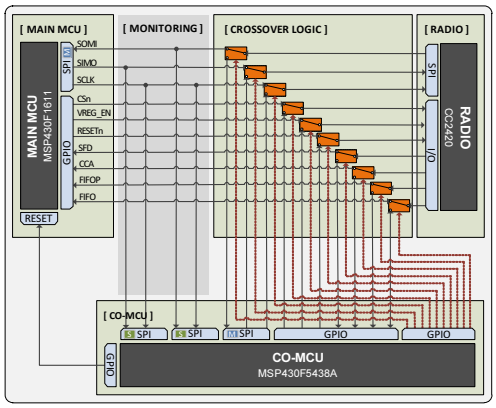


Figure 4: Monitoring and crossover logic configuration.

2.4 Monitoring the Processor-Radio SPI Bus

As mentioned earlier, one of the key features of the SPI-SNOOPER hardware is that it enables the monitoring of the bidirectional communication between the main processor and the radio by transparently snooping the SPI communication bus interface between the main processor and the radio. To do this, the SPI bus lines are forked to two independent SPI interfaces of the reliability co-processor, each of which is configured to act as an SPI slave. One SPI interface on the co-processor (let us call this interface `CoProc_SPI_1`) is dedicated to monitoring the `SIMO` line of the main SPI bus. `SIMO`, in this context, stands for Slave Input Master Output. As the name implies, this line contains a serial bitstream of the data that the main processor (*i.e.*, the bus master) sends to the radio (*i.e.*, the bus slave). This line is connected to the `SIMO` pin of `CoProc_SPI_1`. Now, at each transition of the SPI clock, `CoProc_SPI_1` also receives the data bit that was being sent to the radio. The second SPI interface on the co-processor (let us call this interface `CoProc_SPI_2`) is dedicated to monitoring the `SOMI` line of the main SPI bus. `SOMI`, in this context, stands for Slave Output Master Input. As the name implies, this line contains a serial bitstream of the data that the radio sends to the main processor. This line is connected to the `SIMO` pin of `CoProc_SPI_2`. Now, at each transition of the SPI clock, `CoProc_SPI_2` also receives the data bit that was being received by the main processor from the radio. The other control signals shown in Figure 1 are forked off and connected to GPIO pins of the co-processor. The area marked with "Monitoring" in Figure 4 shows how the processor-radio SPI bus is branched out to the co-processor. `CoProc_SPI_1`, `CoProc_SPI_2`, and the relevant GPIO pins on the co-processor are configured as interrupt-enabled peripherals, thereby allowing the the co-processor to efficiently monitor all the traffic on the SPI bus.

2.5 Crossover Logic

In addition to the bus monitoring ability described above, the reliability co-processor on the SPI-SNOOPER board also has the capability of disconnecting the main processor from the radio and assuming control of the SPI bus itself to communicate directly with the radio. The key component that enables this functionality is

Component	Count	Cost (USD)	
		Unit	Total
MCU	1	4.55	4.55
USB	1	2.95	2.95
SPDT Switch	10	0.13	1.30
PCB	1	8.50	8.50
Misc	-	-	5.00
Total	-	-	22.30

Table 1: Additional cost of one SPI-SNOOPER board (not including components found on the TELOS mote) in quantities of 1000.

the crossover logic on the SPI-SNOOPER board. The crossover logic consists of 10 SN74LVC1G3157 single-pole double-throw (SPDT) switches from TI. The SN74LVC1G3157 SPDT switch provides sufficient switching speed (typically 0.5ns) and operates using a wide supply voltage range (1.65V to 5.5V), while consuming 0.05μA (typical) for maintaining its status regardless of specific supply voltage level [12]. These SPDT switches are controlled by the reliability co-processor, as illustrated in Figure 4. Depending on the control input to the SPDT switches, the switches bridge the radio's SPI and control lines either to the main processor (this mode of operation is illustrated as *Route 1* in Figure 2) or to the co-processor (this mode of operation is illustrated as *Route 1* in Figure 2). The SPI port on the co-processor is configured to be in master mode so that it can initiate bus transfers to the radio if the radio is connected to it.

Note that the reliability co-processor can decide to re-configure the SPDT switches and take control of the radio under various conditions. One common scenario is for the co-processor to make this decision based on the outgoing/incoming data that it sees during passive SPI bus snooping. In such a scenario, perhaps the reception of a specific type of packet (*e.g.*, a special command from the base station) can trigger the co-processor to takeover the radio and communicate directly with the base station.

2.6 Cost Analysis

Table 1 shows the cost of the additional components on the SPI-SNOOPER board, except for the ones included in the TELOS platform, assuming the production of 1,000 boards.

Although SPI-SNOOPER employs additional peripherals such as LEDs and USB for debugging and testing purposes, the co-processor and SPDT switches are the only essential components required for SPI-SNOOPER functionality. According to TI's website, possibly because of the product lineup shift, the reliability co-processor MSP430F5438A costs $4.55 per unit for 1,000 units while the main processor MSP430F1611 costs $8.25 per unit for the same amount (as of March 2012). Thus, the added component cost of transforming a platform such as the TELOS mote in a co-processor augmented one is minimal, further justifying this approach.

3. SPI-SNOOPER SOFTWARE ARCHITECTURE

In order to maintain the transparent nature of SPI-SNOOPER, we make absolutely no modification to the software that runs on the main processor. The only information we need is the type of the operating system it runs. The reason is that different operating systems configure the SPI bus and the CC2420 radio differently (more on this in Section 3.3). Rather than porting an OS, we developed the software that runs on the co-processor from scratch in C to keep it lightweight and efficient. Figure 5 shows the key components of the SPI-SNOOPER software architecture.

SPI-SNOOPER can operate in two different modes:

- *Passive Mode:* In this mode, the co-processor listens to all SPI communication as a slave (Section 2.4) The main purpose of this mode is logging the communication that takes place through the SPI bus. In this mode, the co-processor can also access the sensors for checking the integrity of certain types of data that the main processor sends to other nodes. Based on certain types of events (*e.g.*, if the rate of transmission exceeds a threshold) or commands received from other nodes (*e.g.*, the base station), SPI-SNOOPER can switch to the *active mode*.
- *Active Mode:* In this mode, the co-processor works as a master of the SPI bus. It uses the crossover logic (Section 2.5) to assume the control of the bus and the radio. It can then use the CC2420 radio to communicate with other devices in the network. It can also route packets to maintain connectivity of the network, if necessary.

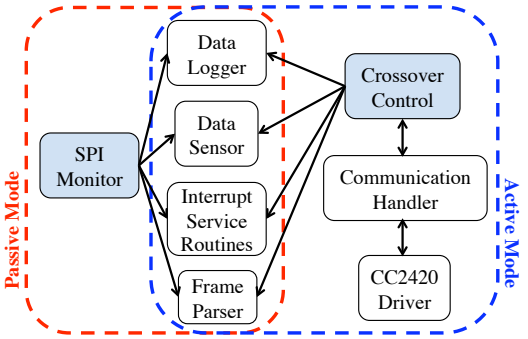


Figure 5: SPI-SNOOPER software architecture

Now, we describe some of the key components of the SPI-SNOOPER software as shown in Figure 5.

- *SPI Monitor:* In passive mode, the co-processor snoops into the SPI bus of the TELOS mote as a slave. The main purpose of this is to be able to identify and record the incoming and the outgoing packets in a mote. This module performs the necessary detection and recording. The details of the detection mechanism is presented in Section 3.2. It can make use of other components in the software stack as well. For example, it often uses the *Data Logger* to log the intercepted bytes to the flash or to the serial port.
- *Crossover Control:* The hardware in SPI-SNOOPER allows the co-processor to assume full control of the SPI bus in TELOS and operate as a master. This control may be necessary in certain situations. For example, if the co-processor suspects malicious behavior by the main processor. This module is used when SPI-SNOOPER operates in active mode. Like the SPI Monitor, it also uses other components of the SPI-SNOOPER software. For example, it may use the *Communication Handler* to communicate with other nodes in the network.
- *Communication Handler:* In active mode, the co-processor is able to and may need to communicate with other nodes. *Communication Handler* performs this task. In our implementation, a CC2420 driver (Figure 5) is provided for the low level communication. It also maintains a routing table if it has to route packets to other nodes.
- *Data Logger:* Sometimes it is expensive or impossible to send the data collected by the co-processor using the CC2420 radio, especially in passive mode. Data Logger uses the onboard flash storage in MSP430F5438A to store the data locally, mostly in meta-data format, which can be retrieved later (Section 4.2). It can also forward the logs to the serial port during run-time. In order to reduce the logging overhead (*e.g.*, write time to the flash), it often logs data in batches.

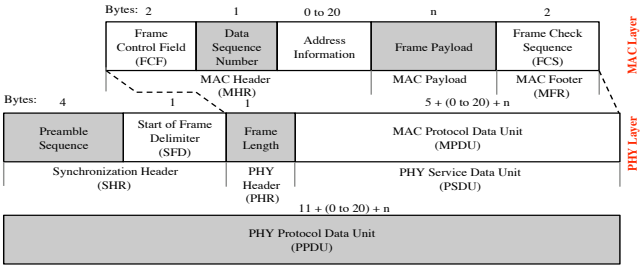


Figure 6: Schematic view of the IEEE 802.15.4 Frame Format

- *Data Sensor:* This module helps to use the onboard sensors in TELOS to be accessed by the co-processor. The samples collected from those sensors can be used to verify the integrity of the data that the main processor is sending (*e.g.*, locally collected light data).
- *Applications:* These are the actual applications (*e.g.*, network logger) that run on the co-processor. As marked with the dotted lines in the Figure 5, an application can work in both active mode and passive mode.

There are also some additional components in the SPI-SNOOPER software architecture, for example, *Frame Parser*, *Interrupt Service Routines*, *etc*. We implemented the SPI-SNOOPER software for TELOS motes running the Contiki [8] operating system.

3.1 Contiki

Contiki [8] is a lightweight operating system developed for resource limited networked embedded systems like wireless sensor networks (WSNs). Contiki contains two communication stacks: uIP [7] and Rime [9]. uIP is a TCP/IP stack for communication using IP and Rime is a lightweight communication stack designed for low-power radios.

3.2 Monitoring Network Communication

The co-processor in SPI-SNOOPER can listen to all communication between the main processor and the radio in TELOS motes. In this section, we will describe how we identify incoming or outgoing packets within the co-processor in SPI-SNOOPER when the main processor is running Contiki.

Detecting Incoming Packets:

Among the pins used for interfacing CC2420 (see section 2.1), The pin `FIFOP` is used to interrupt the microcontroller when a complete frame has been received. As soon as this interrupt is received, co-processor starts recording the bytes that are passed from CC2420 to the main processor (*i.e.*, from `SOMI` in CC2420 to `SOMI` in the main processor). This recording continues until the receive buffer at CC2420 is empty, which can be identified by monitoring the `FIFO` pin. Even though the `FIFOP` pin generated the interrupt when the frame was received at the radio, the main processor may delay the receiving process by few cycles. As a result, just by observing the status of those two pins, we cannot deterministically identify the presence of a packet in the SPI bus. When the processor is ready to receive a packet, it writes `CC2420_RXFIFO | 0x40` (*i.e.*, `0x7F`) to the SPI bus. Henceforth, we refer to this byte as `START_READING`. As a result, whenever the co-processor detects a packet has arrived at the radio, it starts tracking the bytes that are being sent by the main processor. Location of the `START_READING` byte helps us to reliably identify the exact starting point of an incoming packet. Figure 7 shows an example sequence of bytes read by the main processor running Contiki while receiving a packet. As shown there, a packet is read in three parts: the length (red),

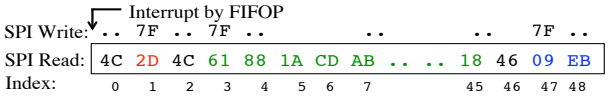


Figure 7: Example of a byte sequence read from the SPI bus while receiving a packet

the payload (green) and the footer (blue). Reading of each part is initiated by sending the `START_READING` byte to the radio.

Detecting Outgoing Packets:

In the previous section, we have shown how two of the pins in CC2420, namely `FIFOP` and `FIFO`, help us to recognize any incoming packets. But for outgoing packets, pins in CC2420 are not very helpful since their statuses are meaningful only when data is available in the radio buffer. That is why we continuously record the bytes written to the SPI bus by the main processor. Before we explain how we detect outgoing packet transmission by monitoring the SPI bus, it is important to understand the high level overview of how Contiki sends a packet using the CC2420 radio. The sending process takes place in two steps: *Prepare* and *Transmit*. During *Prepare*, the MAC header and the MAC payload (Figure 6) are loaded to the radio's transmit buffer. The MAC footer is added by the radio hardware itself. In Contiki, at the beginning of *Prepare*, the processor sends a strobe signal (`CC2420_SFLUSHTX` or `0x09`) to the radio. After that, the packet is sent in two steps: *1)* the frame length (one byte) and *2)* MAC header and the MAC payload (Figure 6). If CRC check is enabled, a two bytes checksum value of the payload is also sent in a third step. Before each step, the address (`CC2420_TXFIFO` or `0x3E`) of the CC2420 transfer buffer is written to the SPI bus. From this, it is easy to recognize that each packet loading should start with the following byte pattern: `0x09 0x3E frame_length 0x3E`. We continuously monitor the outgoing byte stream and as soon as we identify the above pattern, we conclude it as the beginning of a new packet sending process. However, detecting a packet while it was being loaded to the radio's transmit buffer does not indicate that the packet was actually sent from the radio. The actual sending process takes place during the next step: *Transmit*.

During *Transmit*, the processor instructs the radio to start the actual transmission, which may or may not be successful based of the status of the radio. The sequence of bytes that are written to the SPI bus during this phase depends on the configuration of the radio. If the radio is configured to send a packet with *clear channel assessment (CCA)*, the main processor sends two strobe signals: `CC2420_SRXON` (`0x03`) and `CC2420_STXONCCA` (`0x05`). If CCA is not enabled, it only sends `CC2420_STXON` (`0x04`). An arbitrary number of `CC2420_SNOP`s (`0x00`) may also be sent in between the aforementioned signals. To resend an already loaded packet (*e.g.*, if re-transmission is enabled), the processor just needs to issue the above strobe signals again. Thus, multiple *Transmit* steps may follow a single *Prepare* step. After detecting a packet during *Prepare*, SPI-SNOOPER also records the number of times the packet was attempted to send by recording the signals transmitted during one or more occurrences of *Transmit*. Figure 8 shows the sequence of bytes that was sent through the SPI bus while transmitting a packet twice. Please note that the payload length is 43 even though the frame length is 45 (2D in hex). This is because the CC2420 hardware adds two bytes of Frame Control Sequence (Figure 6) before sending a packet.

3.3 SPI-SNOOPER in Other Operating Systems

Fundamentally, the SPI-SNOOPER hardware is capable of working in any sensor network OS, not only Contiki (*e.g.*, TinyOS [4]). But

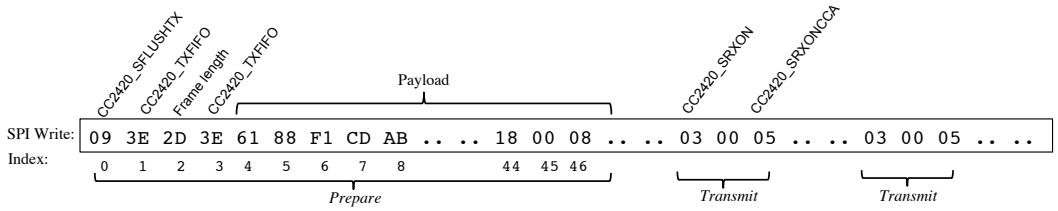


Figure 8: Example of a byte sequence written to the SPI bus while sending a packet

Program	Memory Usage (bytes)	
	RAM	ROM
Contiki (Hello World)	5,396	23,254
monitoring + no logging + no crossover	617	2,448
monitoring + flash logging + no crossover	905	3,124
monitoring + serial logging + no crossover	903	3,180
monitoring + serial logging + crossover	1,210	5,084

Table 2: Memory footprint of SPI-SNOOPER

the same software may not be compatible across different types of operating systems. The reason is that different operating systems handle CC2420 radio communication differently. One example is how incoming packets are handled in TinyOS, where it follows a different configuration than Contiki. In Section 3.2, we explained how we detect incoming packets using the interrupts generated by the `FIFOP` pin in CC2420. By default, `FIFOP` is an active high pin. But in TinyOS, it is configured to be an active low pin. Thus, the same detection mechanism will not work for TinyOS. We can overcome such issues by modifying the detection mechanisms in SPI-SNOOPER software and without modifying anything in the underlying hardware. This is true for other operating systems as well. In this paper, we demonstrate SPI-SNOOPER using the Contiki operating system.

4. EVALUATION

We evaluated SPI-SNOOPER using a set of microbenchmarking experiments and a set of applications. The accuracy of the methods that SPI-SNOOPER uses to detect incoming and outgoing packets along with some other things (*e.g.*, sensing, logging, *etc*.) is shown using these experiments. We demonstrate four use cases where SPI-SNOOPER can be used for transparent monitoring and control of the network communication. The first experiment demonstrates how network communication can be logged and later be used using SPI-SNOOPER. The second application shows that with prior knowledge of the application packets, SPI-SNOOPER can verify the integrity of the sensor readings that are being sent from a mote. In our third application, we use SPI-SNOOPER to detect abnormal behavior of the main processor and take control of the SPI bus to maintain network connectivity. Finally, the fourth example demonstrates how SPI-SNOOPER can open up an emergency backdoor in a mote for communicating with others in the network. All our motes were TELOS with Contiki 2.5 installed. We used the Rime communication stack [9] for network communication.

4.1 Microbenchmarking

The experiments that were used to microbenchmark SPI-SNOOPER are presented in this section.

4.1.1 *Memory Footprint of* SPI-SNOOPER

Table 2 shows the memory footprint of SPI-SNOOPER software in different configurations. With all the features enabled (*e.g.*, crossover logic, sensing) and the ability to log using the serial port, the SPI-SNOOPER software only incurs around 5KB of ROM usage. The RAM footprint also is very small. For the sake of comparison, we also included the memory usage of a basic Contiki program compiled using the default configuration.

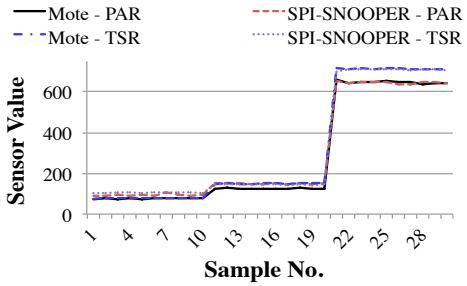


Figure 9: Comparison of the samples collected from the light sensors (PAR and TSR) by SPI-SNOOPER and the associated mote

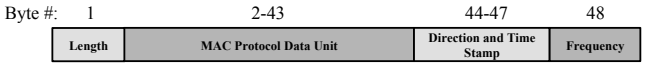
4.1.2 *Accuracy of Intercepting Communication*

Since consecutive bytes of a packet are sent using the SPI bus at a fast speed (*e.g.*, CC2420 supports up to 10MHz SPI clock speed), the co-processor has to operate faster than this speed. This is because it not only has to be able to detect each individual bytes, but it may also have to do some additional processing (*e.g.,* storing the data) between consecutive bytes. Hence, the co-processor is set to operate in a fully interrupt-driven manner while running at maximum speed (25MHz).

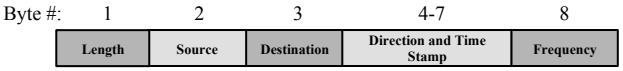
We used two TELOS motes broadcasting 802.15.4 packets. Most of the radio settings were left to the defaults (*e.g.*, transmission power and channel) except we used `nullrdc` for the radio duty cycling (RDC) layer in order to avoid packet re-transmissions. In future experiments, we will show that SPI-SNOOPER also works properly with other RDC layer protocols, such as, `contikimac`. In the first set of experiments, we used the TELOS mote that was being monitored by SPI-SNOOPER for periodically broadcasting packets that contained the word `Hello`. We varied the packet-sending rate from 1-packet/10 seconds to 128 packets/second. In all cases, we were able to detect 100% of the packets as they were being sent. We also varied the packet sizes by changing the payload and achieved the same level of accuracy. In our next set of experiments, we used the other TELOS mote to broadcast similar packets while leaving the mote equipped with SPI-SNOOPER in receive-only mode. We varied the transmission rates similarly and in all cases, we were able to detect all the packets that were received by the mote. In all experiments, we logged the entire packet to the flash along with some additional information (more on this in the next section). The accuracy was determined in a post-processing phase, where we read the logs and checked the data sequence numbers of the packets.

4.1.3 *Accuracy of Sensing by* SPI-SNOOPER

TELOS contains two optional light sensors: `PHOTOSYNTHETIC` (PAR) and `TOTAL_SOLAR` (TSR). In our hardware design, we allow SPI-SNOOPER to access the sensors in a transparent way. Since, both



(a) Log format for logging the entire contents of a packet



(b) Log format for logging the metadata of a packet

Figure 10: Possible log formats in SPI-SNOOPER

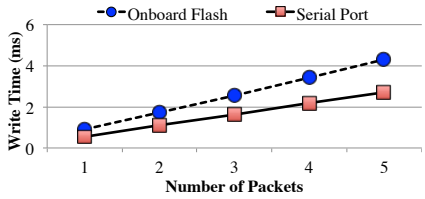


Figure 11: Time needed to log communication to flash and serial port

the main processor and SPI-SNOOPER use the same physical sensors, it is important to verify that the readings they provide are similar and accurate. Figure 9 shows samples collected in three different light conditions by the main processor and SPI-SNOOPER simultaneously (1-sample/5 seconds). As can be seen in the figure, values collected by both of them were similar. Later on, we also show that the values collected by the mote equipped with SPI-SNOOPER are similar to those collected by the regular motes (Section 4.3).

4.1.4 *Overhead of Logging in* SPI-SNOOPER

As we mentioned earlier, SPI-SNOOPER allows us to log the monitored information to the onboard flash of the co-processor. While data is being written to the flash, all interrupts are disabled. In order to reduce the interruption, we write logs in batches. For example, we logged the entire packet contents for the experiments in Section 4.1.2 in batches of 5 logs. Figure 10(a) shows the format of a log entry used in our experiments. Each log entry was 48 bytes long. Besides containing the MAC protocol data unit of a packet (Figure 6), it also contained the length, direction (incoming or outgoing) and timestamp of the packet. The first bit of byte 44 was used to indicate the direction of a packet. Moreover, we also recorded the number of times a packet was transmitted (for outgoing packets) in the last byte of a log entry. This is useful when re-transmissions are enabled. To make the flash writing process efficient, we used the *Block Write* mode [2]. If we have 256K of flash storage available for logging, we can store 5,461 logs using the format shown in Figure 10(a). If we use the more compact format shown in Figure 10(b), we can store up to 32,768 logs.

Figure 11 shows the time needed to perform the log writing operation. As the number of packets increases in a batch, the time required also increases linearly, as shown in the figure. For the sake of comparison, it also shows the timer required to forward logs to the serial port.

4.1.5 *Power consumption*

As we mentioned, the SPI-SNOOPER board also contains components specific to a TELOS mote. Hence, it is difficult to measure the power consumption of only the components that are required for the co-processor to function properly. So, we measured the power consumption of an off-the-shelf TELOS mote and a SPI-SNOOPER board separately. For both cases, the main processor was running the same application that broadcasts 10 packets for 0.5 seconds. Except for that, all the functionality for the SPI-SNOOPER board, such as monitoring and logging, was enabled. As can be seen in the figure 13, the average current consumption was around 18.15 mA for a regular TELOS mote and around 22.75mA for the SPI-SNOOPER board. In all cases, the supply voltage was 3.0 V. It should be noted that the additional current consumption was not only due to the co-processor, but also due to other peripherals such as a number of LEDs and USB ports, which were placed on the SPI-SNOOPER prototype board for testing and debugging convenience. It is possible to reduce the overall power consumption of the SPI-SNOOPER board significantly by removing the non-essential components and putting the co-processor in a low power mode whenever the main processor is in low power mode.

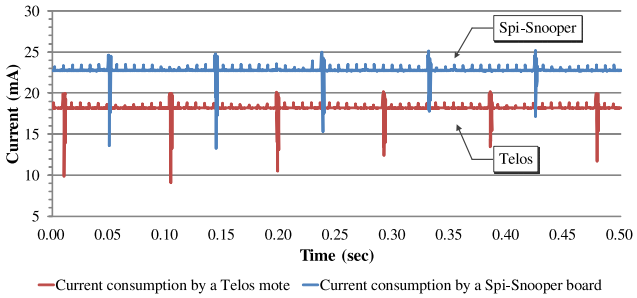
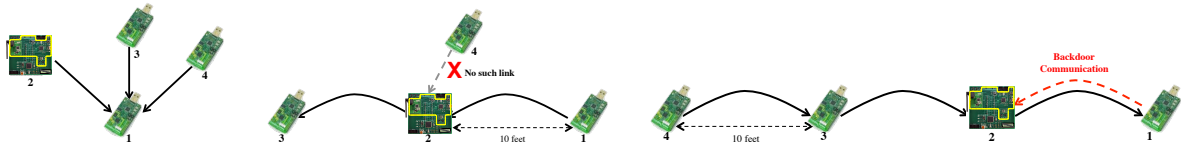


Figure 13: Current consumption by TELOS and SPI-SNOOPER

4.2 Logging Network Communication

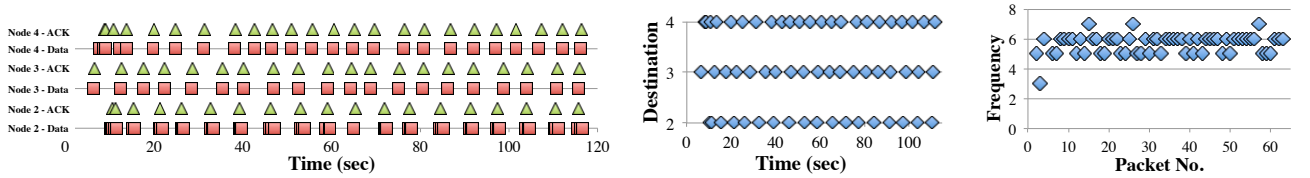
We have already shown how SPI-SNOOPER can be used to log SPI bus communication using the onboard flash or serial port. In this experiment, we show how this information can later be used for analyzing the communication that took place in a mote. We used a network topology similar to the one shown in Figure 12(a) except that the base station (node 1) was equipped with SPI-SNOOPER. The TELOS motes were sending packets to the base station using the mesh routing protocol with the Rime communication stack. Packets were being sent periodically with a random interval of 4 – 6 seconds. We also had `contikimac` in the RDC layer with packet acknowledgements and re-transmissions enabled. Each data packet and acknowledgement packet had a size of 45 bytes and 5 bytes respectively (physical layer). Whenever the base station received a packet, it sent a packet back to the original sender. We logged the entire contents of all incoming and outgoing packets to the flash using the format shown in Figure 10(a). Later on, we retrieved the stored logs from the flash and re-constructed the trace of incoming and outgoing packets to and from node 1. Figure 14(a) shows the timeline for all the packets received. This includes both the data packets and the acknowledgements. We constructed this timeline from the timestamps generated by SPI-SNOOPER. Figure 14(b) shows similar traces for the outgoing packets. A point in the graph means an attempt to send a packet that is already in the buffer of the radio was made (Section 3.2). Note that only outgoing data packets are shown since the radio was configured to send acknowledgements for incoming packets automatically. Since re-transmissions were enabled, multiple attempts were made to send a packet until acknowledgements were received or timeouts occurred. Figure 14(c) shows the frequency of such attempts to send each packet from the base station.

In Figure 15, we show a trace produced from the logs generated by SPI-SNOOPER, where two motes were broadcasting packets with extremely high frequency (64 packets/second). Only one mote was equipped with SPI-SNOOPER. Each packet was of size 21 bytes in the physical layer. Since each log was time stamped by SPI-SNOOPER,



(a) Integrity verification (b) Handling abnormal behavior (c) Providing an emergency backdoor

Figure 12: Topologies used for the experiments



(a) Incoming data packets and acknowledgements (b) Outgoing data packets (c) Frequency of outgoing packets

Figure 14: Trace of incoming and outgoing packets generated from the logs collected by SPI-SNOOPER

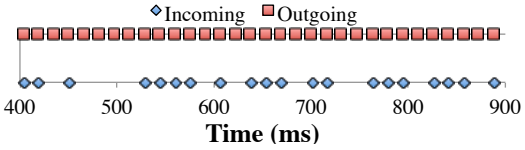


Figure 15: Trace generated from logs generated by SPI-SNOOPER for high frequency packet transmission between two nodes

we were able to draw this timeline with this high resolution. The figure shows that the number of incoming packets was less than the number of outgoing transmission attempts. The reason was that many packets did not arrive due to high rate of collision. These experiments demonstrate how low-level details about communication can be gathered because of the level of visibility SPI-SNOOPER provides which otherwise would have been difficult to generate.

4.3 Integrity Verification

With prior knowledge of the application data packets, SPI-SNOOPER can be used to verify the integrity of the sent data. For this experiment, we used the setup shown in Figure 12(a). Motes were running `sky-collect`, a program that comes with the standard Contiki distribution. Motes running this program periodically collect some sensor data (*e.g.*, light, temperature, *etc*.) and forward them to the base station (node 1) using a tree routing protocol. In this experiment, we set the period to 4 – 6 seconds and used `nullmac` in the RDC layer. Each application packet was of size 101 bytes (physical layer). As we have mentioned earlier, SPI-SNOOPER has access to the two light sensors (PAR and TSR) present in a TELOS mote. In this experiment, we equipped node 2 with SPI-SNOOPER. Hence, SPI-SNOOPER could analyze the outgoing packets of node 2; check the values of the light sensors independently by accessing the same set of sensors that the mote used to collect the values. We maintained a history (in this case, the last 5 values for each sensor) of the recently sent values and compared them with the values that were collected by SPI-SNOOPER. If all values sent within a window differed by more than 20% from their corresponding values collected by SPI-SNOOPER, we concluded that the TELOS was malfunctioning and we disconnected it from the radio. We assumed that the sensors were working properly. To emulate this behavior, we intentionally programmed node 2 in such a way that after some time, it started multiplying the collected values by 2 before sending them. As can be seen in Figure 16, motes started sending the sensed values for the two light sensors in normal indoor light

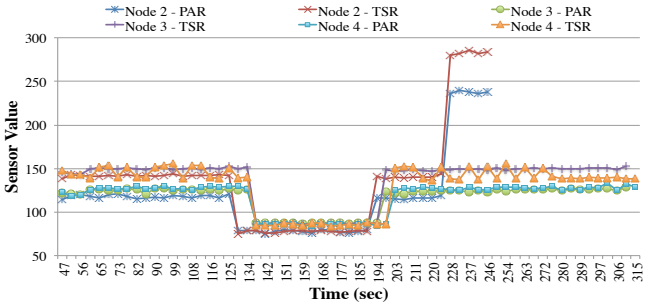


Figure 16: Average light intensity values received by the base station in the experiment described in Section 4.3

condition. We kept our lights off for the time period of 125 – 190 second. The sensed values from all motes in Figure 16 also corroborate that. Around the 220th second, mote 2 started sending values that were significantly higher than the other motes even though all motes were nearby. SPI-SNOOPER was able to determine that these values were different than the actual sensor values. After some time, the main processor in node 2 was disconnected from the radio. In the next experiment, we show how SPI-SNOOPER can keep the communication alive with other motes in the network even after the main processor is disconnected. In this case, we could have used SPI-SNOOPER to keep sending sensor values even after detecting the erroneous readings from the main processor.

4.4 Handling Abnormal Behavior

SPI-SNOOPER can analyze incoming and outgoing packets and determine certain types of malicious/buggy activities by the main processor. We demonstrate one such example here. The experimental setup is shown in Figure 12(b). Node 1 was sending packets to node 3 every 4 – 6 seconds using mesh routing protocol from the Rime communication stack. Since there was no direct connection between them, packets were being routed through node 2, which was equipped with SPI-SNOOPER. As shown in Figure 17, after some time node 3 started receiving packets originating from node 4. However, there were no incoming packets from node 4 to node 2. Hence, this could be due to a bug in node 2 or a malicious behavior (*e.g.*, a wormhole attack) by that node. SPI-SNOOPER noticed that no packets were received by node 2 from node 4, but still it was forwarding packets that seemed to have originated from there. After monitoring this behavior for some time (in this case, after detecting 5 such incidents), it took control of the SPI bus using the

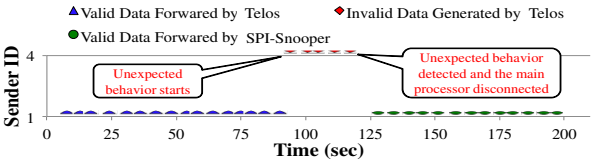


Figure 17: Packets received by node 3 in the Figure 12(b)

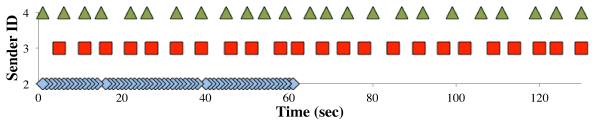


Figure 18: Packets received at the base station in Figure 12(c)

crossover logic (Section 2.5). However, to maintain connectivity of the network, the co-processor kept forwarding the packets that were being received by the radio in node 2 and was not addressed to itself. Before forwarding, the sender address was changed to node 2. Also, it kept a history of packet forwarding by the main processor. Hence, it had a similar routing table available that was used to set the immediate destination address of a packet. Some other information (*e.g.*, hop count) were updated as well. For this, SPI-SNOOPER needed to know what RDC layer protocols were being used in the network as different protocols use different MAC payload header formats. In this case, we used `nullrdc`. Thus, SPI-SNOOPER not only prevented other nodes from the malicious or buggy activity of node 2, but also kept providing services that were essential to maintain the connectivity of the network. Figure 17 shows how node 3 kept receiving packets from node 1 without any interruption. This experiment demonstrates how SPI-SNOOPER can provide emergency services even when the main processor is unavailable or disconnected.

4.5 Providing an Emergency Backdoor

SPI-SNOOPER provides a convenient way to communicate with itself from the rest of the network. Such a backdoor allows other motes, for example, the base station, to send commands or queries to SPI-SNOOPER. We demonstrate one such use case here. We used the network topology in Figure 12(c). As shown there, motes were sending packets to the base station (node 1) using the mesh routing protocol from the Rime communication stack. We used `nullrdc` for the RDC layer. Node 2 was equipped with SPI-SNOOPER. As shown in Figure 18, packets from mote 3 and 4 were being sent with a period of $4 - 6$ seconds. Node 2 was sending packet at a much faster rate (1 packet/second). After about a minute, the base station decided not to receive any packet from node 2. It then sent an instruction to disconnect the main processor in node 2 but to keep forwarding packets from the other motes just like the previous example. SPI-SNOOPER responded to the request by disconnecting the main processor from the radio at node 2. As shown in the figure, no more packets from node 2 were delivered to the base station. But, the base station still kept receiving packets from the other motes. We used the regular application packet to send the "disconnect and forward" instruction to SPI-SNOOPER. However, we may have multiple nodes equipped with SPI-SNOOPER. To address the right SPI-SNOOPER in the right mote, we set the original source address of the application packet containing the instruction to the address of the target mote and added an offset to this. For example, in this case, we added 100. SPI-SNOOPER at node 2 detected a packet coming from 102 and understood that it was a packet addressed to itself containing some instructions from the base station. The assumption was that there was no node in the network with the address 102. We can set the offset to a large number to ensure this. Thus, SPI-SNOOPER provided a backdoor entry to node 2 for the base station. In this approach, the application at node 2 also received the command packet since it was a regular application packet. For communicating using Rime, this is the best way to avoid unnecessary flooding in the network. We can encrypt the packets that contain such commands so that the application running at the mote will not be able to understand the actual meaning of the packet. We can further enhance the solution by using uIP [7] and use a dedicated port number for such communications. That way, the regular applications will not receive these packets anymore.

5. RELATED WORK

The concept of using co-processors or watchdog processors has been prevalent in the traditional computers and high-performance systems for decades [18, 17, 14]. Most of the time, they are used to provide more efficiency (*e.g.*, a graphics co-processor) or an additional layer of security (*e.g.*, a cryptographic co-processor) to the main processor(s). [16] is a survey of concurrent system-level error detection techniques using a watchdog processor in the context of traditional PC based systems. None of these solutions were designed to work with networked embedded systems like WSNs. There are few works that are applicable for WSNs. FlashBox [6] is a hybrid hardware-software solution where an additional MCU is used to log the interrupts that take place in the main processor. For logging to work, applications have to be compiled using a modified version of `avr-gcc`. A more recent work, Aveksha [21], is another hardware-software approach that allows tmonitoring and recording of the internal state of the processor in TELOS motes using the on-chip debug module. However, unlike Aveksha, which only monitors and records the internal states of the processor, SPI-SNOOPER not only passively monitors network communication but can also actively monitor the health status of other components of the node using the information gathered through network monitoring, sensor data integrity checking, *etc.*, to enable reliable and robust operation.

There are few other works [20, 25, 15, 5] that aim to record or log certain types of events at run time and replay them later on. Most of them are either software-only approaches or require pre- or post-processing of the gathered execution trace. For example, LiveNet [10] suggested the concept of a Deployment Support Network (DSN), where the authors just connected two individual TELOS nodes (target node and monitoring node) via wires in order to communicate and implicitly exchange certain data of interest by using software routines that reside on both the target node and the monitoring node. Therefore, LiveNet is not transparent and efficient in terms of energy and cost.

Our solution, SPI-SNOOPER, uses a co-processor based design that enables monitoring the SPI bus of resource constrained wireless sensor devices. Moreover, using the crossover logic, SPI-SNOOPER enables the co-processor to assume the control of the network communication, thus providing reliability and fault tolerance.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented SPI-SNOOPER, a co-processor augmented hardware-software approach that can monitor and control the SPI bus communication between the main processor and the radio in a mote. We have designed and developed a prototype for TELOS [19] motes running Contiki [8] and demonstrated how it works in the real world. In principle, it is not confined to TELOS and applicable to any other wireless sensor node platforms regardless of the type of the main processor because the proposed scheme is nothing to do with certain type of the main processor and only related to the exposedness of SPI bus between the processor and

the radio. In addition, it works with any other operating systems that uses SPI bus for communication between the main processor and the radio. SPI-SNOOPER works in a transparent manner without the knowledge of the processor or applications that run on the accompanying mote. It provides an added layer of in-situ visibility, which enables fine-grained monitoring of the behavior of a mote from the point of view of network communication. Such visibilities are crucial for successful deployment and debugging of networked embedded systems like WSNs. In our experiments, we have shown that SPI-SNOOPER can detect incoming and outgoing packets with 100% accuracy even when communication takes place at a very high rate. With prior knowledge of the application packets, SPI-SNOOPER is capable of verifying the integrity of outgoing packets containing certain types of data (*e.g.*, sensor readings), thanks to its ability to access the TELOS sensors independently. We have also demonstrated how SPI-SNOOPER can provide emergency services, for example, packet forwarding, in the event when the main processor needs to be disconnected. However, there are still scopes of improvement in our design. We have shown how the snooped information can be logged using the 256KB onboard flash of the MSP430F5438A processor. We are exploring the option of adding an additional external and removable flash with much higher capacity. This will significantly increase the flexibility of using the logging feature of SPI-SNOOPER. We are also working on adding a new feature to SPI-SNOOPER, which will allow us to re-program the associated mote using a golden image or a firmware received over-the-air. This will be extremely useful to perform recovery and diagnosis, if necessary. In our current implementation, the co-processor in SPI-SNOOPER has access to the two light sensors present in TELOS. In future, we would like to add the capability of monitoring and accessing other types of sensors and peripherals. We have shown how SPI-SNOOPER can take control of the SPI bus and keep communicating with the other motes in the network. In addition to this, we are interested in emulating the behavior of a radio using the co-processor. This will allow us to implement a true traffic gatekeeping solution like [11], but with more transparency and no dependency on the main processor of the mote.

7. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation (NSF) under grants CNS-0953468 and ECCS-0925851.

8. REFERENCES

[1] List of wireless sensor nodes (Wikipedia). http://en.wikipedia.org/wiki/List_of_wireless_sensor_nodes.

[2] MSP430x5xx/MSP430x6xx Family User's Guide. Texas Instruments.

[3] Telos (Rev. B) Schematics. http://webs.cs.berkeley.edu/tos/hardware/telos/telos-revb-2004-09-27.pdf.

[4] TinyOS. http://www.tinyos.net.

[5] B. Chen, G. Peterson, G. Mainland, and M. Welsh. Livenet: Using passive monitoring to reconstruct sensor network dynamics. *Distributed Computing in Sensor Systems*, pages 79–98, 2008.

[6] S. Choudhuri and T. Givargis. FlashBox: a system for logging non-deterministic events in deployed embedded systems. In *SAC*, pages 1676–1682, 2009.

[7] A. Dunkels. Full TCP/IP for 8 Bit Architectures. In *MobiSys*, May 2003.

[8] A. Dunkels, B. Gronvall, and T. Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *EmNets '04*, 2004.

[9] A. Dunkels, F. Österlind, and Z. He. An adaptive communication architecture for wireless sensor networks. In *SenSys*, pages 335–349, 2007.

[10] M. Dyer, J. Beutel, T. Kalt, P. Oehen, L. Thiele, K. Martin, and P. Blum. Deployment support network. *Wireless Sensor Networks*, pages 195–211, 2007.

[11] M. Hossain and V. Raghunathan. Aegis: A lightweight firewall for wireless sensor networks. In *DCOSS*, pages 258–272, 2010.

[12] T. Instruments. SN74LVC1G3157; Single- Pole Double-Throw Analog Switch (Rev. G). http://www.ti.com/lit/ds/symlink/sn74lvc1g3157.pdf, 2008.

[13] N. Kothari, K. Nagaraja, V. Raghunathan, F. Sultan, and S. Chakradhar. HERMES: A Software Architecture for Visibility and Control in Wireless Sensor Network Deployments. In *IPSN*, pages 395–406, 2008.

[14] D. Lu. Watchdog processors and structural integrity checking. *IEEE Transactions on Computers*, 100(7):681–685, 1982.

[15] L. Luo, T. He, G. Zhou, L. Gu, T. Abdelzaher, and J. Stankovic. Achieving repeatability of asynchronous events in wireless sensor networks with envirolog. In *INFOCOM*, pages 1–14, 2006.

[16] A. Mahmood and E. McCluskey. Concurrent error detection using watchdog processors– a survey. *IEEE Transactions on Computers*, pages 160–174, 1988.

[17] I. Majzik, W. Hohl, A. Pataricza, and V. Sieh. Multiprocessor checking using watchdog processors. *Computer Systems Science and Engineering*, 11(5):301–310, 1996.

[18] A. Pataricza, I. Majzik, W. Hohl, and J. Hönig. Watchdog processors in parallel systems. *Microprocessing and Microprogramming*, 39(2-5):69–74, 1993.

[19] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *IPSN*, pages 48–es, 2005.

[20] V. Sundaram, P. Eugster, and X. Zhang. Efficient diagnostic tracing for wireless sensor networks. In *SenSys*, pages 169–182, 2010.

[21] M. Tancreti, M. S. Hossain, S. Bagchi, and V. Raghunathan. Aveksha: A hardware-software approach for non-intrusive tracing and profiling of wireless embedded systems. In *SenSys*, 2011.

[22] Texas Instruments. MSP430F15x, MSP430F16x, MSP430F161x Mixed Signal Microcontroller (Rev. G) . http://www.ti.com/lit/ds/symlink/msp430f1611.pdf.

[23] Texas Instruments. MSP430 Hardware Tools User's Guide. http://www.ti.com/lit/ug/slau278h/slau278h.pdf, 2010.

[24] Texas Instruments. MSP430F543xA, MSP430F541xA Mixed Signal Microcontroller (Rev. B). http://www.ti.com/lit/ds/symlink/msp430f5438a.pdf, 2010.

[25] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson. Replay debugging of real-time systems using time machines. In *IPDPS*, pages 8–pp. IEEE, 2003.

[26] K. Whitehouse and et al. Marionette: Using RPC for Interactive Development and Debugging of Wireless Embedded Networks. In *IPSN*, 2006.

[27] J. Yang, M. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *SenSys*, pages 189–203, 2007.