Software Verification for TinyOS

Doina Bucur
University of Oxford, UK
doina.bucur@comlab.ox.ac.uk

Marta Kwiatkowska
University of Oxford, UK
marta.kwiatkowska@comlab.ox.ac.uk

ABSTRACT

Ensuring the reliability of the software deployed on networked wireless sensors is a difficult problem: unsafe, low-level, interrupt-driven code runs without memory protection in dynamic environments. To aid the matter, we describe a software analysis tool for the *debugging and verification* of TinyOS 2, MSP430 applications at *compile-time*. While existing solutions act at runtime to log, report and reboot from software errors such as memory violations, our tool is the first to allow the programmer to verify a TinyOS application statically; given assumptions about the behaviour of the node's environment and *assertions* upon the state of the node itself, the tool explores all possible program executions and returns to the programmer an error trace leading to the violation of an assertion, if any exists. Besides memory-related errors (out-of-bounds arrays, null-pointer dereferences), we also support application-specific assertions, including low-level assertions upon the state of the registers and peripherals.

Categories and Subject Descriptors

I.2.9 [**Artificial Intelligence**]: Robotics—*Sensors*; D.4.5 [**Operating Systems**]: Reliability—*Verification*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

Keywords

wireless sensor networks, TinyOS, reliability, debugging, software verification, model checking, bounded model checking, tos2cprover, CBMC, CProver

1. INTRODUCTION

Reliable sensor software is difficult to program. Interrupt-driven code runs on embedded operating systems without user/kernel boundary, memory management or protection, and the code follows a low-level programming paradigm with unrestricted access to the microcontroller's mapped memory. Writing an accidentally null pointer in a TinyOS [12] application running on a TelosB, MSP430-based platform [19], i.e.

```
struct sensor_data {
  nx_uint16_t seq_no;
  nx_uint16_t data[10];
} *sensor_data;          // pointer not initialized

// write into sensor_data->data[9]
read_sensor(sensirionSht11_temperature_read,
          sensor_data->data[9]);
```

writes the output register `P1OUT` of peripheral port P1 (mapped starting at address `0x0020`), potentially modifying the operation of those modules controlled through the `P1OUT` bits [21]. Likewise, an out-of-bounds array access on the stack may corrupt not only adjacent stack memory, but also other sections of the memory map.

With software errors this destructive, the programmer's difficult task lies with ensuring that no such violations remain in the code by deployment time, or that they can be recovered from. The task is complicated if the sensor node is due to function in a dynamic, error-prone network: the programmer has to both account for any context switch from the kernel thread to an incoming interrupt handler, and for any—including corrupted—incoming data e.g., received on the radio interface. The advent of the TOSThreads [13] API for programming applications with abstracted blocking threads (instead of nesC [10] interrupt handlers) only partially aids the situation: the bulk of the code deployed on a mote will still have been initially programmed in interrupt-driven nesC.

Three trends of research are answering the problem of programming reliable sensor applications: One develops tools for *runtime monitoring and recovering* from error, e.g., Safe TinyOS [9] and Neutron [5]; A second, including the present work, aims at error detection at *compile-time*; The third parts with the well-used, existing sensor languages and operating systems, and develops new high-level, strongly typed languages and programming tools for sensors, such as Virgil [22], to eliminate some of the programmer's responsibilities.

We focus on *compile-time debugging* for the mainstream TinyOS operating system, and motivate this focus by noting that, thus far, the task of locating programming errors in a sensor application traditionally takes the form of *debugging by deployment*, in which case staring at a set of blinking LEDs allows little visibility into the fault's cause. Runtime recovery tools aid the debugging process by providing the error's location (and in the best of cases, the program trace which has led there). With such tools, however, error detection is at its most expensive: for each detected error, the deployed application has to suffer through i.e., a node's reboot, the standard error-correcting measure. Furthermore, software faults may even persist though reboots, or the deployment environment which triggers them may be difficult to guess or reproduce.

In turn, we provide the first method to allow the programmer to debug a TinyOS application without deploying it. In the same way an application is usually *simulated* pre-deployment to show that it adheres to an expected functionality (e.g., in a real-code simulator such as TOSSIM [18]), we allow TinyOS code to be instrumented with *assertions* which should hold whenever they are reached, and input the resulting code into a fully-automated *verification* toolchain which returns a program trace leading to a violation of an assertion, if any such violation exists.

The difference between simulation and verification is that between *may* and *must*: a program's simulation unwinds only one of all possible program traces, and thus may or may not make visible a desired property of the program; a program's verification, on the other hand, will unwind all possible program executions, translates the resulting state machine and any assertions into formal, mathematical models (e.g., Boolean formulas), and checks whether the conjunction of the program's formula and a assertion's negated formula is satisfiable—in which case, there exists a trace on which the assertion is violated.

Our toolchain closes the link between TinyOS software and the state of the art in verification tools for standard C. A programmer's nesC or TOSThreads application and the existing TinyOS code database—with added assertions—are passed through the nesC compiler, `nescc` [10], and the platform-specific, monolithic code resulted is inputted to `tos2cprover`, our source-to-source transformation tool. In turn, `tos2cprover` automatically gives standard C semantics to all of the code's platform-specific features, and instruments the code with calls to interrupt handlers, thus mimicking incoming interrupt requests (*IRQ*s) from the hardware. Consequently, the resulting C code serves as input to CBMC [6], a software verification tool for ANSI C; CBMC finally instruments the code with another set of assertions guarding against standard memory violations and exceptions, and boundedly unwinds the program's loops, reporting assertions' violations occuring within the unwinding limit.

The difficulty of our verification method is two-fold. First, giving high-level C semantics to low-level microcontroller flavours of C requires detailed knowledge of the microcontroller's workings. Second and more importantly, the verification procedure only scales if the size of the program's state-space is minimized; to this end, (i) *partial order reduction* is used when instrumenting the application with calls to IRQ handlers, to reduce the number of calls to be considered, and (ii) the *amount of unwinding* is carefully set only large enough to exhibit the required assertion violations.

As formal verification is generally not *complete* for infinite-state programs (i.e., cannot unwind and verify the entire state-space), we intend our automated verification toolchain (presented in Section 3) not to replace, but to complement runtime methods: we show (in Section 4) that important bugs can be automatically localized with our bounded verification technique before deployment time, thus reducing the frequency of the expensive runtime recovery procedures. The scalability of the method and its time costs are also given in Section 4.

2. BACKGROUND

2.1 TinyOS

TinyOS (in the form of its two major releases, TinyOS 1 [12] and 2 [17]) is the mainstream operating system for wireless sensor network devices. NesC [10] components, including the programmer's application, are wired together to form an OS-wide program; nesC is designed under the expectation that a final, inlined code will be generated from components by whole-program compilers, a fact which allows for better code generation and analysis.

All lengthy commands in TinyOS (e.g., the sending of a packet on the radio) are non-blocking; their completion is signalled by an *event* (some of which are triggered by a hardware interrupt), whose handler should be brief, instead posting *tasks* to the system's task queue for further execution. All threads of control in a TinyOS application are thus rooted in either an event handler or a task, in a two-level concurrency model: event handlers run with highest priority and preempt the lower-priority tasks, which execute most of the program logic. Tasks run to completion, however, and *synchronous code* is that which is only reachable from tasks; *asynchronous code* is reachable from at least one event handler. Whenever program variables are accessible to both synchronous and asynchronous code, a potential data race ensues.

TOSThreads [13] is TinyOS 2.x's recent thread library. It preserves the system's existing concurrency model, and adds a lowest third priority level for the new, *application threads*. The threads are now allowed blocking system calls, which execute as regular TinyOS tasks, and which unblock the application thread when completed. As such, TOSThreads provide the traditional, POSIX-threads-like programming paradigm for TinyOS, can be programmed in either nesC or C, and come complete with synchronization methods.

2.2 Sensor platforms

TinyOS serves a variety of hardware platforms. Telos [19] motes are based on the 16-bit Texas Instruments MSP430 [21] microcontroller; Mica nodes are built around Atmel's AVR [3], and Intel's 8051 [2] power e.g., MITes nodes. `nescc`, the nesC compiler, wires, inlines and translates the TinyOS components into platform-specific, non-standard C; for an MSP430 platform, this language is that of `mspgcc` [23], a port of the GNU tools to the Texas Instruments MSP430 microcontrollers. What distinguishes the language from standard C is its extensions for directly utilizing hardware: the fragment of code

```
static volatile uint8_t r __asm ("0x0031");
r &= ~(1 << 6);
```

or in other words

```
*(volatile uint8_t *)49U &= ~(1 << 6);
```

clears a bit in the 8-bit peripheral register `P5OUT` at location `0x0031`, where the three LEDs are memory-mapped on a MSP430 TelosB; this amounts to turning the yellow LED on. Similarly, a function declared with the attributes

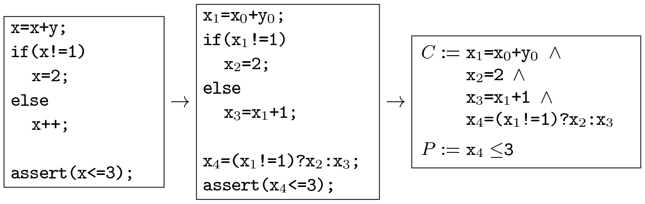
```
__attribute((wakeup)) __attribute((interrupt(14)))
```

declares that `sig_ADC_VECTOR` is an interrupt service routine for interrupt line 14, and that it wakes the processor from any low power state as the routine exits.

2.3 CBMC—A bounded model checker

The state of the art in software verification tools includes highly usable tools for the analysis of ANSI-C programs; CBMC [6] (part of the CProver [15] tool suite for the formal verification of both hardware designs—e.g., programmed in Verilog—and software programs) uses bounded model checking techniques [7] to verify against the violation of assertions in ANSI-C code such as that expected from safety-critical, embedded systems. These programs are—more so than high-level applications—a challenge to debug, due to the extensive use of pointers, pointer arithmetic, and bit-wise operators.

To derive an accurate mathematical representation of an input program, CBMC first translates the code into `goto-cc` [15], a simpler, intermediate representation of ANSI-C, for which (i) all side-effect assignments are broken into equivalent statements by introducing auxiliary variables, (ii) all loops and recursive function calls are *unwound* by a user-provided number of times, by duplicating the loop body, (iii) function calls are inlined, and (iv) static analysis techniques optimize the code by constant propagation. The resulting program is then transformed (using a pointer analysis in the process) into *static single assignment form* (SSA), a standard intermediate representation in which every variable is split into "versions", i.e., a new variable is invented for each assignment to the original, as exemplified in Fig. 1; frequently used for compiler optimizations, the technique simplifies the analysis of the variables' definition and use.



```
x=x+y;
if(x!=1)
  x=2;
else
  x++;

assert(x<=3);
```

→

$x_1 = x_0 + y_0;$

$\text{if}(x_1 \mathrel{!}= 1)$

$\quad x_2 = 2;$

else

$\quad x_3 = x_1 + 1;$

$x_4 = (x_1 \mathrel{!}= 1) ? x_2 : x_3;$

$\text{assert}(x_4 <= 3);$

→

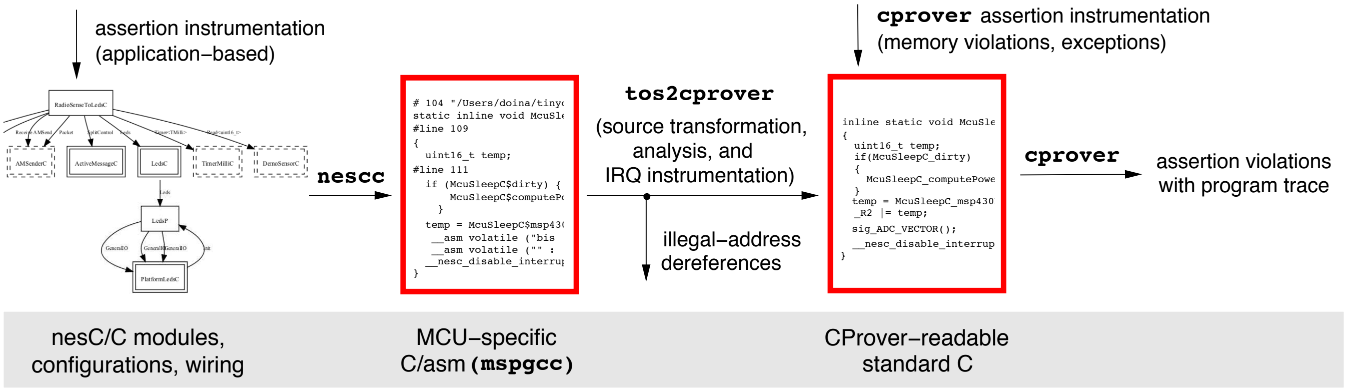
$$C := x_1 = x_0 + y_0 \;\wedge\; x_2 = 2 \;\wedge\; x_3 = x_1 + 1 \;\wedge\; x_4 = (x_1 \mathrel{!}= 1) ? x_2 : x_3$$

$$P := x_4 \leq 3$$

Figure 1: CBMC program transformation into a mathematical model

The procedure then precisely represents ANSI-C data types, pointers, structures and arrays to the bit level (with the word-width adjustable, i.e., to 16 bits, by a command-line option), instruments them with with assertions checking e.g., array bounds and arithmetic exceptions, and finally produces two boolean propositional formulas: C for the program itself, and P for the asserted expression, as in Fig. 1. P is then verified by converting $C \wedge \neg P$ into conjunctive normal form (CNF) and then passing it to a SAT solver such as MiniSAT [20]. If this conjuncted formula is satisfiable, there exists a violation of the assertion, and CBMC returns to the programmer a program trace leading to the violation, as a debugging tool would do; otherwise, the assertion holds.

In verifying an ANSI-C program by bounded unwinding, CBMC proves a partial guarantee of program properties (i.e., that bugs are absent for a certain amount of unwinding), is highly automated and scales reasonably well.



```
# 104 "/Users/doina/tiny
static inline void McuSl
#line 109
{
  uint16_t temp;
#line 111
  if (McuSleepC$dirty) {
      McuSleepC$computeP
    }
  temp = McuSleepC$msp43
   __asm volatile ("bis
   __asm volatile ("" :
  __nesc_disable_interru
}
```

```
inline static void McuSle
{
  uint16_t temp;
  if(McuSleepC_dirty)
  {
    McuSleepC_computePowe
  }
  temp = McuSleepC_msp430
  _R2  |= temp;
  sig_ADC_VECTOR();
  __nesc_disable_interrup
}
```

Figure 2: The verification toolchain

3. STATIC TINYOS SOFTWARE SAFETY

In TinyOS, software bugs stem from both the legacy nesC component base (the lowest levels of which are platform-dependant), and the programmer application's components. By a *safe* TinyOS program, we understand that which exhibits no memory violations, and whose programmer-inserted assertions hold.

To achieve a homogeneous verification scheme for both legacy and newly programmed TinyOS code, we created an automated toolchain of program transformation and verification as depicted in Fig. 2. In order to be able to treat both nesC components and the C/nesC TOSThreads identically, an initial run of TinyOS's `nescc` compiler is used to generate an inlined, platform-specific, low-level C program; instead of employing the platform's own compiler to further build this into a binary deployable on a mote, the program is passed to our own tool, `tos2cprover`, which has a double task:

- In a source-to-source transformation step, it gives ANSI-C meaning to all low-level, hardware-managing extensions, and instruments the code so as to mimick the hardware's functionality: whenever a register's value is filled in from the hardware, the program is augmented so as to provide such values.

- Then, `tos2cprover` reads the functions' attributes and determines which functions would be called as IRQ handlers in the event of a hardware interrupt, then instruments the resulting program so that IRQ handlers are called whenever hardware interrupts are allowed. A reachability analysis step is used to minimize the occurence of such calls.

Following these transformation and instrumentation steps, the result is a high-level, standard C program which precisely (yet minimally) preserves the functionality of the initial platform-specific program and its hardware.

3.1 Tos2CProver: source-to-source transformation

Table 1 exemplifies the source transformations executed by `tos2cprover` on a TelosB, MSP430 program. While `mspgcc` code implicitly assumes an underlying memory map in which low, constant addresses have a semantics (e.g., writing at `0x0031` programs the LEDs), this memory is replaced with a header file defining global replacement variables; e.g., `_P5OUT` is now the 8-bit output register for peripheral port 5. All subsequent dereferences of address `0x0031` are replaced into accesses to `_P5OUT`[1]. As a note, the Status Register `_R2` has the General Interrupt Enable (GIE) as bit 4; if GIE is set, interrupts are enabled.

Then, `mspgcc`'s assembly extensions are straightforwardly translated into standard C, as is all other non-standard language (e.g., identifier names are standardized by replacing dollar signs with underscores, struct and union designated initializers are expanded).

3.2 Tos2CProver: IRQ instrumentation

The `nescc`-generated program inputted to `tos2cprover` doesn't explicitly call any IRQ handlers; in deployments, the calls are made from the hardware. Instead, it defines the functions and marks them as interrupt service routines; e.g., in the case of a TelosB-based *Sense*, two types of hardware interrupts are expected: one from the user timer, TimerB, and another from the 12-bit Analog-to-Digital Converter, ADC:

```
void sig_TIMERB1_VECTOR(void)
 __attribute((wakeup)) __attribute((interrupt(24)));
void sig_ADC_VECTOR(void)
 __attribute((wakeup)) __attribute((interrupt(14)));
```

Thus, *asynchronous code* in *Sense* is that which is rooted (i.e., reachable from) either of the two IRQ handlers,

[1] Note that the variables we introduce are names in accordance to the MSP430 documentation [21], but are preceded by an underscore, to avoid name clashes with existing program variables.

Table 1: tos2cprover: source-to-source transformation examples for MSP430 code.

<table>
<tr><th>MSP430-specific program feature</th><th>Example tos2cprover transformation</th></tr>
<tr><td>MCU registers and memory map</td><td>Standard C global variables
<code>unsigned short _R2;   /* Status Reg */</code>
<code>unsigned char _P5OUT;   /* 0x0031 P5 */</code>
<code>unsigned short _ADC12CTL0;   /* 0x01A0 ADC12 */</code>
<code>unsigned short _ADC12MEM[16]; /* 0x0140 ADC12 */</code></td></tr>
<tr><td>Fixed-address dereference
<code>*(uint8_t*)49U &= ~(0x01 << 6);</code></td><td>Global variable access
<code>_P5OUT &= ~(0x01 << 6);</code></td></tr>
<tr><td>Fixedly allocated variables
<code>uint16_t HplAdc12P$ADC12CTL0 __asm ("0x01A0");</code>
<code>HplAdc12P$ADC12CTL0 |= 0x0010;</code></td><td>Global variable access

<code>_ADC12CTL0 |= 0x0010;</code></td></tr>
<tr><td>Assembly instructions
<code>__asm volatile ("eint");</code>
<code>__asm volatile ("bis %0, r2" : : "m"(temp));</code></td><td>C instructions
<code>_R2 &= 0x0008;</code>
<code>_R2 |= temp;</code></td></tr>
</table>

while *synchronous code* is rooted (and only reachable from) the `main` function. The size of the asynchronous code is substantial: take e.g. *Sense*, in which out of the 520 reachable functions in the program, 166 are reachable from the `ADC` interrupt handler and 185 from the `TIMERB0` handler; 386 (both synchronous and asynchronous functions) are reachable from `main`.

To simulate the presence of interrupts, tos2cprover needs to instrument the program with explicit, atomic calls to the handlers of the expected hardware interrupts, i.e., `sig_ADC_VECTOR()`, with each call guarded by a check of the GIE bit, and each call made atomic by disabling and enabling interrupts (as TinyOS events always run to completion):

```
/* IRQ INSTRUMENTATION */
if (int_enabled()) {    /* if (_R2 & 0x0008) */
  disable_int();        /* _R2 &= ~0x0008;   */
  sig_ADC_VECTOR();
  enable_int();         /* _R2 |= 0x0008;    */
}
```

A correct, yet naive, approach is to instrument the program by refactoring it to use threads, and running the IRQ handlers as separate threads in parallel with a `main` thread (or, equivalently, adding calls as every second statement in all `main`-reachable code). However, each instrumented IRQ call amounts to the duplication of all code rooted in the call. We employ two fully-automated minimization procedures to reduce the number of these instrumentations, as follows.

The first is a *partial order reduction* technique [8], a general method to reduce the number of interleavings between threads by exploiting the fact that a number of different interleavings have the same end effect to the main thread, and thus need only be checked once. E.g. for the standard TinyOS `main` (selected lines; the comments are ours):

```
    int main (void)
    {
       // interrupts are disabled to start with
 1:    RealMainP_Scheduler_init();
 2:    RealMainP_PlatformInit_init();
 3:    RealMainP_SoftwareInit_init();
       // and finally enabled
 4:    __nesc_enable_interrupt();

 5:    RealMainP_Scheduler_taskLoop();
    }
```

adding our guarded IRQ calls before any of the lines 1-4 will not describe a realistic interrupt scenario, as our `int_enabled()` guard will always evaluate to false, and the IRQ handler itself will never be run. Similarly, unless a statement contains a potential *data race* (i.e., data is accessed asynchronously—a classical example in TinyOS being the task queue, `uint8_t SchedulerBasicP_m_next[SchedulerBasicP_NUM_TASKS]`, as events post tasks), an IRQ call *before* the statement has the same effect as one *after* the statement.

More formally, at this minimization stage tos2cprover calculates the set of global program variables which are accessed asynchronously, and instruments the program with the following reduced set of IRQ calls:

- A call for e.g. `sig_ADC_VECTOR()` appears before each statement containing a read of a variable raced between the `ADC` interrupt and `main`. This is sound, but overapproximated: the statement may execute in an atomic context, in which case the call isn't reachable.

- A call also appears (i) before the beginning of those atomic sections where a data race may happen, or (ii) in the MCU's interruptable sleep.

This first minimization step leaves e.g., in the case of

Sense, a total number of 91 IRQ calls between the two types of IRQs.

A second, stronger analysis step is then used to minimize this result; to remove the overapproximation above, tos2cprover runs CBMC to check for the reachability of each of the 91 instrumentations: the verification of an assert(0); inserted in the body of the interrupt routine fails when the assertion is reachable. This step leaves *Sense* with a manageable, minimum set of 8 IRQ calls.

3.3 Instrumentation with assertions, nondeterminism and assumptions

Our toolchain verifies, for each inputted program, two types of assertions. For both the existing TinyOS code base and any new applications, assertions can be manually inserted by the programmer and are preserved as such in the transformed program source. These *application-based assertions* can be either hardware-aware, e.g.

```
assert(_P5OUT & 0x0008);
```

or high-level, (e.g., asserting upon the value of a variable local to a component).

Furthermore, CBMC automatically inserts *memory-violation assertions* guarding both bounds of all array accesses, null-pointer dereferences, and other exceptions such as arithmetic division by zero. As function SchedulerBasicP_pushTask(uint8_t id) writes upon the task queue:

```
SchedulerBasicP_m_next[SchedulerBasicP_m_tail]=id;
```

with id unsigned, CBMC will generate the upper-bound assertion:

```
Claim SchedulerBasicP_pushTask.1:
  array ‘SchedulerBasicP_m_next’ upper bound
  (unsigned int)SchedulerBasicP_m_tail < 8
```

For *Sense*, 132 memory-violation assertions are thus generated[2]. Advantageously, this generation is completely automatic, with CBMC analysing an array's declaration to find the index bounds; SafeTinyOS [9], for example, has programmers explicitly type-annotate arrays with access bounds.

Finally, a decision needs to be taken in regard to the contents of those registers and buffers whose values are filled in by the hardware and not the software. For example, reading the current time in a TinyOS application takes the form of reading the user timer's count register, _TBR (mapped at address 0x0190), which holds the number of clock periods elapsed since the last timer interrupt, and which is automatically incremented from the hardware at every clock period. In another example, the 8-bit _U0RXBUF buffer (mapped at 0x0076) holds the latest byte received from the network. A similar discussion holds for setting TOS_NODE_ID, the variable which holds the node's address, and which is programmed at deployment time.

Clearly, the actual values in such registers drive the program's further behaviour. We set their values in either of two ways:

- The register envolved is assigned a nondeterministic (i.e., *any*) value, and the verification procedure explores all the ensuing possibilities.
- The register is *assumed* to have a particular value, or to have any value within a small, particular set.

3.4 CBMC and program unwinding

As a final step in our toolchain, the transformed program annotated with assertions is passed to CBMC configured for 16-bit words, and each claim is verified at a time, for scalability. The runs need to have specified an unwinding depth; this can be either the identical for all loops and recursions, or—ideally—selectively refined for each. Some of the loops are obviously of fixed iterations; e.g., out of the 16 loops from *Sense*, the loop:

```
do {
  *resultBuffer++ =
    Msp430Adc12ImplP_HplAdc12_getMem(i);
}
while(++i < length);
```

is always bounded at 16 iterations (the size of the ADC12 conversion memory)[3]. Other loops, on the other hand, are clearly unbounded, such as the main OS scheduler loop in function SchedulerBasicP_Scheduler_taskLoop. For our benchmarks in the following, we make a visual inspection of the program's loops (as reported by CBMC), determine bounds for the loops which are clearly bounded, and experiment with unwinding depths for the rest. For the purpose of determining the reachability of instrumented IRQ calls (described in Section 3.2), we set the depth for the unbounded loops to the minimum, 1.

4. THE BENEFITS AND COSTS OF VERIFICATION

For our tests, we settle on the existing applications in the apps directory from TinyOS's source tree; we pick applications which wire TelosB components of different functionality, as summarized in Table 2.

We detail the size and complexity of the test cases in terms of (i) the lines of code in the cleanly reformatted

[2] The names of functions, variables and assertion identifiers in all our code examples are those generated by nescc: the original nesc function or variable name is preceded by a list of nesc component names, which helps in recovering the C code's correspondent in the original nesc code.

[3] A few loops are bounded, but not worthy of exploring, and are thus commented out. An example is the initial clock calibration, which busy waits for thousands of clock periods.

Table 2: TelosB-based test cases

	Blink	*Sense*	*TestDissemination*
functionality lines of code, number of loops memory-violation assertions	timer 3340, 8 35	sensor, timer 7181, 16 132	CC2420 radio, timer 13388, 31 747
expected interrupts	`TIMERB0`	`TIMERB0`, `ADC`	`TIMERB0`, `PORT1`, `PORT2`, `UART0RX`, `UART0TX`
reachable functions	total: 248, `TIMERB0`: 114	total: 520, `TIMERB0`: 185, `ADC`: 166	total: 1022, `TIMERB0`: 364, `PORT1`: 153, `PORT2`: 25, `UART0RX`: 268, `UART0TX`: 16
potentially raced global variables	`TIMERB0`: 6	`TIMERB0`: 7, `ADC`: 11	`TIMERB0`: 15, `PORT1`: 13, `PORT2`: 0, `UART0RX`: 19, `UART0TX`: 0
IRQ instrumentations	initial 21, minimized to 4	initial 92, minimized to 8	initial 422, minimized to 30

program outputted by `tos2cprover`, (ii) the number of unique loops for which CBMC needs to have configured an unwinding depth, (iii) the number and type of expected hardware interrupts, together with qualitative measures of the size of the code duplication incurred during the IRQ instrumentation phase. As a side note, the program generated by `nescc` and inputted to `toscprover` is not completely optimized; for our test cases, this input program contained code of no end functionality, such as that rooted in the IRQ handlers for the non-user timer, `TIMERA0/1`; `tos2cprover` skips instrumenting the program with such IRQ calls. On another hand, for *TestDissemination* we preserved the code for the `UART0RX/TX` interrupts, and instrumented the program with the respective calls: UART functionality may not be wired through to the top application component, but supports the CC2420 radio.

Finally, Table 2 gives the precise number of IRQ instrumentations calculated as in Section 3.2, and the number of automatically generated, memory-violation assertions; most of the assertions are array bounds checks, with a number of null-pointer dereference checks.

In the remaining of this section, we give an overview of our verification runs, discuss their scalability, advantages and limitations.

4.1 Out-of-bounds array access, null-pointer dereference, and application-based assertions

We ran our MSP430 test cases through the verification toolchain, having set to *any value* the contents of the TimerB count register `_TBR`, the 16 ADC12 sensor memory buffers `_ADC12MEM[]`, and the transmit and receive buffers `_U0TXBUF/_U0RXBUF`.

Any verification run is parametrized by the following measures:

- The number of IRQ calls added per code rooted in a single iteration of the scheduler main loop. While `tos2cprover` calculates the program points at which an IRQ of a certain type can be called, a verification run may include all, none, or any superset of these calls. This is settled empirically on a per-application basis: one `TIMERB0` interrupt is sufficient to explore the workings of *Blink*, and similarly for `ADC` and *Sense*; for any network communication, on the other hand, an interrupt arrives for any byte received, which induced us to allow more `UART0` transmit or receive interrupts per loop.
- The number of main loops which CBMC unwinds in the `SchedulerBasicP_Scheduler_taskLoop` method. Given some understanding of the task loop functionality, and the number of IRQ calls per loop, we again settle the number empirically, per-application.
- The number of assertions checked in one verification run; this number can be either *one* (and CBMC is configured with the assertion's identifier) or *all*; as checking one assertion at a time scales better, we automatized our tool to iterate through all of the program's assertions.

All our verification runs of the memory violations in the test cases from Table 2 came up negative, when allowed two task loops and one IRQ per loop (in the case of *Blink* and *Sense*) and up to eight loop and eight IRQs per loop for *TestDissemination.* We then artificially triggered some positive runs in *TestDissemination.* First, we sent a null pointer to a `requestData` call in the `DisseminationEngineImplP` module from TinyOS's network library (the comments are ours, and long function names are wrapped on multiple lines):

```
static void
  DisseminationEngineImplP_sendObject(uint16_t key)
```

```
{
  void *object;
  uint8_t objectSize = 0;
  [..]
  // send a zero instead of &objectSize
  object = DisseminationEngineImplP_
           DisseminationCache_requestData(key, 0);
}
```

Since the `requestData` method does no sanity check on the pointer it receives:

```
inline static void *
  DisseminatorP_0_DisseminationCache_requestData
  (uint8_t *size)
{
  *size = sizeof(DisseminatorP_0_t);
  [..]
}
```

the assertion then generated by CBMC to check the sanity of the pointer:

```
Claim DisseminatorP_0_DisseminationCache_requestData.1:
  line 7503 function
    DisseminatorP_0_DisseminationCache_requestData
  dereference failure: NULL pointer
  !(SAME-OBJECT(size, NULL))
```

fails. Second, as a means to make visible such incorrect parameter passing even when sanity checks are in place, we cause a large `objectSize` being passed to the `send` call:

```
static void
  DisseminationEngineImplP_sendObject(uint16_t key)
{
  void *object;
  uint8_t objectSize = 0;

  // we cause this to set objectSize to a large value
  object = DisseminationEngineImplP_
           DisseminationCache_requestData
           (key, &objectSize);

  // which is then sent to a send call
  DisseminationEngineImplP_AMSend_send(
    AM_BROADCAST_ADDR,
    &DisseminationEngineImplP_m_buf,
    sizeof(dissemination_message_t) + objectSize);
}
```

We then introduce an `assert(0)` to strengthen the sanity check in the implementation of `send`:

```
static error_t CC2420ActiveMessageP_AMSend_send(
  [..] uint8_t len)
{
  if(len >
     CC2420ActiveMessageP_Packet_maxPayloadLength())
  {
    assert(0);
    return ESIZE;
  }
}
```

Whenever `ESIZE` would be returned, the assertion is now also reachable and reports failure, and a program trace to track the location of the faulty call.

4.2 Constant-address dereference

A potential, secondary source of errors in embedded software is that of dereferencing constant memory addresses. While null pointers can still be erroneous (as exemplified by Section 4.1), dereferencing constant, low pointers is generally expected from embedded code. To enforce a degree of *safety* when dereferencing constants is envolved, we state that all dereferencing of constants must be limited to constants from those memory-map sections which pertain to peripheral control (I/O locations), and not to other sections.

To this end, in the process of program transformation, `tos2cprover` reports to the programmer the list of encountered memory dereferences, and translates the constant address implicated to its section in the memory map, e.g., for the line:

```
*(volatile uint8_t *)49U ^= 0x01 << 6;
```

we report

```
-> DEREF at 49/0x31 in the 8-bit Peripheral Module
        in *(volatile uint8_t *)(49U)
```

and for a fixedly allocated variable:

```
static volatile uint8_t r __asm ("0x0019");
r |= 1 << 1;
```

we report

```
-> DEREF at 25/0x19 in the 8-bit Peripheral Module
        with fixed-address variable r
```

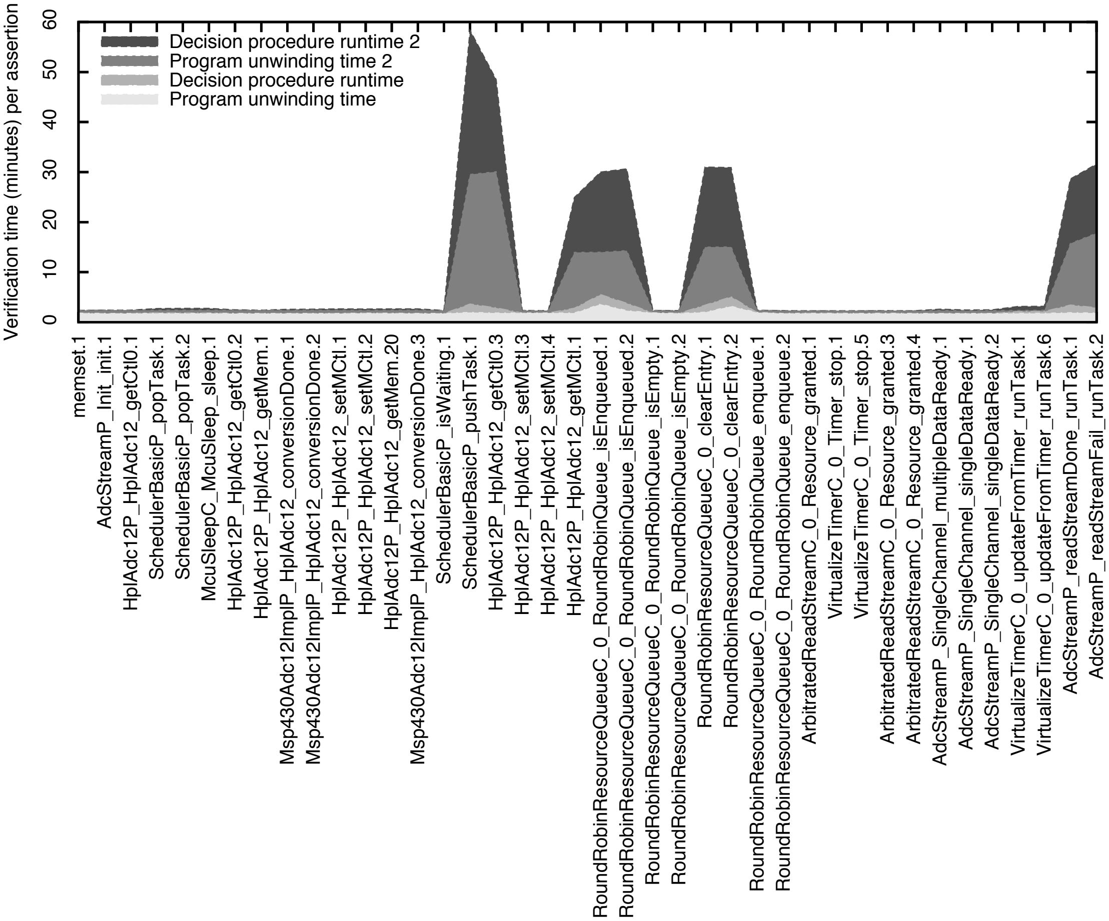
In some cases (particularly for dereferences of address `0x0`), an inspection of this report is advisable to sort any null pointers from legitimate peripheral access. A similar approach is taken by SafeTinyOS [9], which has programmers explicitly mark legal dereferences of constants with a *trusted type*, and thus make null-pointer derefences visible.

4.3 Verification time

The cost of showing that a TinyOS application is safe lies partially in (i) inspecting the program's loops to settle on an unwinding bound for each, (ii) inspecting the list of expected hardware interrupts to decide on the number of IRQ instrumentations necessary, and (iii) inspecting the report on dereferencing contant addresses. Mostly, however, the cost lies with the verification time: the time it takes the model checker to unwind the program (i.e., the *program unwinding time*), and generate its mathematical formula and have this verified by the SAT solver (i.e., the *decision procedure runtime*).

Fig. 3 exemplifies the verification times for a respresentative subset of memory-violation assertions from *Sense*. The x axis is labeled with identifiers of assertions: e.g., `SchedulerBasicP_pushTask.1` is the first assertion generated regarding code in function `SchedulerBasicP_pushTask`. When more than two assertions

Figure 3: Verification times for selected memory-violation assertions in *Sense*



are generated for a single function, we note that verification times are similar for all these assertions, and only depict the first and the last. Two verification runs are given for each assertion (each run in terms of both program unwinding time and of decision procedure runtime); both runs are configured with one IRQ call per task loop; the first run unwinds the task loop once, and the second twice.

We note that most assertions are verified in a speedy manner, using close to zero time in the decision procedure, and a constant time for program unwinding. There are, however, notable exceptions for which the verification time explodes—we used these time-consuming runs to bound CBMC's unwinding depth: for e.g. *Test-Dissemination*, some five-task-loop verification runs took up to 55 minutes.

5. RELATED WORK

5.1 Runtime error protection

Most existing solutions against software errors in sensor operating systems act at *runtime*: the code is instrumented such that a statement which is semantically *unsafe* under its current execution context is detected before it is executed, and one or another diagnosis or recovery measure is taken (which usually consists in reporting the error and rebooting, as summarized in Table 3). While a necessary solution to ensure safe execution in all execution contexts, runtime error detection is necessarily followed by the expensive redeployment of already-deployed sensor software, and could instead be preceded by static error detection to save on redeployment efforts.

Safe TinyOS [9] detects memory and type violations in deployed, running code, and transfers control to a fault handler, which either reboots or powers down after sending a concise failure report to its base station. The failure report identifies the error type and its source code location (but doesn't give an execution trace clarifying the context which caused the error); the failure report is used to debug the code post-deployment. While the method allows the *safe execution* of existing TinyOS code, with little programmer effort and runtime overhead, debugging every error encountered envolves the software's redeployment.

A drawback of Safe TinyOS's Deputy code instru-

Table 3: Runtime diagnosis and recovery solutions to software errors

Scheme	OS	Scope of errors	Measure
Safe TinyOS, *Neutron*	TinyOS	out-of-bounds array access, invalid-pointer dereference, integer-to-pointer cast	report error location, reboot; *Neutron*: with state preservation
Interface contracts	TinyOS	pre-/postconditions for `nesC` interface use	LED-signal error location
NodeMD	MantisOS	stack overflow, deadlock, livelock, application assertions	report execution trace, remote queries

menter is the fact that, unlike our automatic instrumentation with assertions, SafeTinyOS programmers must explicitly type-annotate e.g., a `void *payload` array with access bounds `COUNT(len)`. The resulting program is first translated into C by the nesC compiler, and then by Deputy into a program instrumented with calls to a failure routine `if(i >= len) deputy_fail()`. Statements accessing a fixed address such as `0x0031` (classic memory violations on desktop systems, but standard in TinyOS) also need manual *trusted cast* annotations, `TC()`. Like SafeTinyOS though (and unlike earlier SafeTinyOS versions), we do not change the C data representation in the verification process.

Neutron [5] adds a welcome update to Safe TinyOS, for applications coded in TinyOS's TosThreads API: instead of rebooting the node as a corrective measure to a memory violation, Neutron groups the application's threads into recovery groups, and selectively restarts the threads in certain recovery units. Furthermore, it allows the preservation of the values held in "precious" memory locations between thread restarts.

The *interface contracts* [1] write specification-like type annotations which define the "correct" use of TinyOS 1.x nesC component interfaces, just as Safe TinyOS annotates memory for safe use. In cases when the semantics of the interface dictates it being stateful, the contract is expressed in terms of the state of the interface: for the interface command `Timer.start()`, the contract states as precondition the fact that the timer's state must be `IDLE`, and as a postcondition that the state of the timer changes (i.e., to `ONE_SHOT`) if the command's return value is `SUCCESS`. In other cases, the interface is stateless and the pre- and postconditions are imposed upon the command's arguments, which can themselves become stateful. This detects incorrect ordering of interface commands at runtime, e.g., a `SendMsg.send()` with a message buffer for which no `SendMsg.sendDone()` event was received to complete a previous send.

NodeMD [16] detects runtime errors in MantisOS multithreaded, synchronous code. AVR-based applications are instrumented to check the stack pointer `SP` for overflows at function-call time, and for the violation of application-specific assertions. The checks for deadlock and livelock involve an artificially added timer to check that each thread goes through its duty cycle.

5.2 Compile-time error detection and simulation

A degree of compile-time verification is reported by [4]. While its scope is limited to TOSThreads applications written in C, it avoids some of the costs of system-wide verification by writing models for the interfaces to system calls (in the style of the runtime interface contracts [1]). Calling `amRadioReceive(&msg, ...)` from the application is modelled so that it preserves its original behaviour: the call returns any of a set of error codes, and `msg` receives (possibly nondeterministic) data. Then, the method verifies the programmer's application on its own; the errors it focuses on are those pertaining to interface use, and application-specific assertions.

FSMGen [14] takes another approach to error detection in TinyOS programs: it statically analyzes the program and derives automatically a finite-state machine to describe the high-level application logic, thus aiding the programmer's understanding of the application code.

TinyOS's own nesC compiler has a basic built-in *datarace* detector, which warns when a global variable is updated from non-`atomic` asynchronous code without having been explicitly tagged with `norace`. While writing `atomic` asynchronous code is good practice for `nesC`, failure to do so only potentially causes a race, as programmers may have used other syncronization idioms (i.e., guards on variables). A suitable context model for race checking [11] then gave an algorithm for the elimination of such false positives.

A weaker case of static error detection is simulation. While it does not prove any guarantee on the program's behaviour, simulation allows for error detection and is particularly realistic in the case of e.g., TOSSIM [18], which accurately simulates TinyOS applications from their own implementation.

6. CONCLUSIONS AND FUTURE WORK

Our work on statically checking safety violations at compile-time is motivated by a general idea that—while an inherently unsafe language like C is the inevitable choice to program embedded systems in—measures must be taken to increase the safety of such programs to a certain, guaranteed, standard. While static verification can never be complete for an infinite-state program, even a low depth of program unwinding can expose memory-violation bugs which would have otherwise made a sensor node fail in the field, post-deployment.

We see our future work as two-fold. First, since our current test cases covered only a small subset of the existing TinyOS code base, we intend to extend our tool and verification tests to (i) other MSP430-specific or higher-level components, (ii) the AVR platform, and (iii) TinyOS applications which are currently deployed in pervasive healthcare applications, where software safety is at its most important. Second, we will look towards a *compositional* means of verifying TinyOS code, so that the verification of assertions which are local to a component may be reused between applications.

7. REFERENCES

[1] W. Archer, P. Levis, and J. Regehr. Interface contracts for TinyOS. In *Proceedings of the international conference on Information Processing in Sensor Networks (IPSN)*, pages 158–165. ACM, 2007.

[2] Atmel. Atmel 8051 microcontrollers hardware manual. `www.atmel.com,doc4316.pdf`, 2008.

[3] Atmel. 8-bit AVR microcontroller with 128K bytes in-system programmable Flash. `www.atmel.com,doc2467.pdf`, 2009.

[4] D. Bucur and M. Kwiatkowska. Bug-free sensors: The automatic verification of context-aware TinyOS applications. In *Proceedings of the European conference on Ambient Intelligence (AmI)*, volume LNCS 5859, pages 101–105. Springer Verlag, 2009.

[5] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr. Surviving sensor network software faults. In *Proceedings of the Symposium on Operating System Principles (SOSP)*. ACM, 2009.

[6] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[7] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[8] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[9] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for TinyOS. In *Proceedings of the conference on Embedded Networked Sensor Systems (SenSys)*, pages 205–218. ACM, 2007.

[10] D. Gay, P. Levis, and R. von Behren. The nesC language: A holistic approach to networked embedded systems. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 1–11. ACM, 2003.

[11] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *Proceedings of the conference on Programming Language Design and Implementation (PLDI)*, pages 1–13. ACM Press, 2004.

[12] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, 2000.

[13] K. Klues, C.-J. Liang, J. Paek, R. Musăloiu, R. Govindan, A. Terzis, and P. Levis. TOSThreads: Safe and Non-Invasive Preemption in TinyOS. In *Proceedings of the ACM conference on Embedded Networked Sensor Systems (SenSys)*. ACM, 2009.

[14] N. Kothari, T. Millstein, and R. Govindan. Deriving state machines from TinyOS programs using symbolic execution. In *Proceedings of the international conference on Information Processing in Sensor Networks (IPSN)*, pages 271–282. IEEE, 2008.

[15] D. Kroening. Formal verification. `http://www.cprover.org/`.

[16] V. Krunic, E. Trumpler, and R. Han. NodeMD: Diagnosing node-level faults in remote wireless sensor systems. In *Proceedings of the international conference on Mobile Systems, Applications and Services (MobiSys)*, pages 43–56. ACM, 2007.

[17] P. Levis, D. Gay, V. Handziski, J.-H. Hauer, B. Greenstein, M. Turon, J. Hui, K. Klues, C. Sharp, R. Szewczyk, J. Polastre, P. Buonadonna, L. Nachman, G. Tolle, D. Culler, and A. Wolisz. T2: A second generation OS for embedded sensor networks. Technical Report TKN-05-007, Technische Universität Berlin, 2005.

[18] P. Levis, N. Lee, M. Welsh, and D. E. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proceedings of the ACM conference on Embedded Networked Sensor Systems (SenSys)*, pages 126–137, 2003.

[19] Moteiv Corporation. Telos. Ultra low power IEEE 802.15.4 compliant wireless sensor module. Revision B : Humidity, Light, and Temperature sensors with USB. `http://www.moteiv.com`, 2004.

[20] N. Sörensson and N. Eén. MiniSat — a SAT

solver with conflict-clause minimization. In *Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 3569 of *LNCS*. Springer-Verlag, 2005.

[21] Texas Instruments. MSP430x1xx family — user's guide (Rev. F). `www.ti.com,slau049f.pdf`, 2006.

[22] B. L. Titzer. Virgil: Objects on the head of a pin. In *Proceedings of the ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 191–208. ACM, 2006.

[23] S. Underwood. Mspgcc—A port of the GNU tools to the Texas Instruments MSP430 microcontrollers. `http://mspgcc.sourceforge.net/manual`, 2003.