Efficient Memory Safety for TinyOS

Nathan Cooprider Will Archer Eric Eide David Gay[†] John Regehr

University of Utah, School of Computing [†]Intel Research, Berkeley

{coop, warcher, eeide, regehr}@cs.utah.edu david.e.gay@intel.com

Abstract

Reliable sensor network software is difficult to create: applications are concurrent and distributed, hardware-based memory protection is unavailable, and severe resource constraints necessitate the use of unsafe, low-level languages. Our work improves this situation by providing efficient memory and type safety for TinyOS 2 applications running on the Mica2, MicaZ, and TelosB platforms. Safe execution ensures that array and pointer errors are caught before they can corrupt RAM. Our contributions include showing that aggressive optimizations can make safe execution practical in terms of resource usage; developing a technique for efficiently enforcing safety under interrupt-driven concurrency; extending the nesC language and compiler to support safety annotations; finding previously unknown bugs in TinyOS; and, finally, showing that safety can be exploited to increase the availability of sensor networks applications even when memory errors are left unfixed.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: *real-time and embedded systems*; C.4 [**Performance of Systems**]: *performance attributes*; D.4.5 [**Operating Systems**]: Reliability; D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.5 [**Software Engineering**]: Testing and Debugging—*error handling and recovery*

General Terms

Languages, Performance, Reliability

Keywords

wireless sensor networks, type safety, memory safety, nesC, TinyOS, Deputy, cXprop, Safe TinyOS

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SenSys'07, November 6–9, 2007, Sydney, Australia.
Copyright 2007 ACM 1-59593-763-6/07/0011 ...$5.00

1 Introduction

Imagine that you have deployed tens or hundreds of wirelessly networked sensors and are using them to collect data. To get to this point, your team has written thousands of lines of interrupt-driven nesC code, which

- runs on motes with just a few kB of RAM, and without a user-kernel boundary or memory protection;
- must cope with highly dynamic situations in which other nodes are failing, rebooting, and going in and out of radio range; and,
- cannot rely on useful abstractions such as blocking threads or a heap.

Given these difficulties, errors such as null pointer dereferences, out-of-bounds array accesses, and misuse of union types are difficult to avoid. Now consider two scenarios.

In the first, a memory safety error in your application corrupts RAM on the faulting node. Such coding errors can have surprisingly hard-to-trace effects, especially on the AVR-based Mica2 and MicaZ motes where the processor's registers are mapped into the bottom of the address space. For example, if p is defined as

```
struct { char x[28]; int y; } *p;
```

and then dereferenced while null

```
p = NULL;
... many lines of code ...
p->y = z;
```

the AVR processor's registers 28 and 29 are modified to contain the low and high bytes of z's value, respectively. Because this pair of registers is heavily used as a memory index, further memory corruption is likely to result.

In general, the behavior of a corrupt node is Byzantine. With many buggy nodes over time, one would expect sensor data to be corrupted, false network routes to be advertised, secret keys to be revealed, and so on. Depending on circumstances, the node may recover, crash, or continue in a faulty manner. It can even be difficult to distinguish between failures induced by software bugs and those caused by hardware-related problems. However, whereas hardware faults can be fixed by swapping out defective parts, software faults persistently degrade the effectiveness of a sensor network. Time is consequently lost pointing fingers, manually rebooting nodes, and staring at code.

In the second scenario, a run-time check detects the impending memory error just before it happens and control is transferred to a fault handler. Depending on how the node is configured, it either reboots or powers down after sending a failure report to its base station. The failure report is concise—a small integer—but serves to uniquely identify the specific error that occurred as well as its location in the original source code. The effectiveness of the sensor network may be degraded until the code can be debugged and redeployed, but in the meantime, the bug has been located and its effects contained.

The first scenario above characterizes the kind of problems that sensor network application developers currently face. The goal of our research is to enable the second scenario by implementing *safe execution* for sensor network applications, in a way that is *practical* given the realities of sensor network software development. Practicality imposes three important constraints. First, we must make existing code safe rather than requiring developers to reimplement applications in a new language or OS. Second, safety must be cheap in terms of programmer effort. Third, because resources are precious on microcontrollers, the run-time overhead of safety must be small.

Our approach to efficient, backward-compatible, safe execution for sensor network nodes is *Safe TinyOS*. "Regular" TinyOS is a popular and open-source platform for sensor network software [12, 18]. A TinyOS-based application is written in nesC [9], a component-oriented but unsafe dialect of C. Our Safe TinyOS platform builds atop regular TinyOS and ensures that programs execute safely, meaning that they respect both type safety and memory safety. A type-safe program cannot conflate types, e.g., treat an integer as a pointer. A memory-safe program cannot access out-of-bounds storage. Together, these properties keep memory errors from cascading into random consequences.

Because safety cannot be fully guaranteed when a program is compiled (in general), the Safe TinyOS toolchain inserts checks into application code to ensure safety at run time. When a check detects that safety is about to be violated, code inserted by Safe TinyOS takes remedial action. In short, Safe TinyOS creates a "red line" [1]—a boundary between trusted and untrusted code—that separates an application running on a sensor node from the small Safe TinyOS kernel that takes control when an application misbehaves.

Our red line enforces safety properties as described above, and the Safe TinyOS toolchain uses cXprop [4]—our powerful static analyzer for embedded C code—to minimize the run-time costs of the red line within Safe TinyOS programs. The measured cost of this safety is low in comparison to the state of practice: i.e., the resources used by unsafe programs as compiled by the regular TinyOS toolchain. One can also apply our cXprop tool by itself to optimize *unsafe* TinyOS programs, and we present data for this scenario to put the costs of Safe TinyOS in greater perspective.

In summary, this paper makes two primary contributions. The first is a detailed presentation of Safe TinyOS, addressing key research challenges in implementing safe execution of sensor network software in a resource-efficient manner. The second is to show that Safe TinyOS is a *practical* system for the development of reliable sensor network software. Notably:

	Average change in resource use	
	vs. regular TinyOS toolchain	vs. TinyOS tools + cXprop
ROM (code)	+13%	+26%
RAM (data)	–2.3%	+0.22%
CPU (duty cycle)	+5.2%	+17%

Table 1: Summary of the costs incurred by Safe TinyOS applications. Each percentage shows the average change in resource use for Safe TinyOS programs relative to the corresponding unsafe TinyOS programs (as produced by the default TinyOS toolchain) or relative to the corresponding unsafe TinyOS programs optimized by cXprop.

- *(Section 3.)* Safe execution can be implemented efficiently, even in the presence of interrupt-driven concurrency. Safe TinyOS efficiently and effectively reports the location of run-time type and memory safety failures.
- *(Section 4.)* The resource costs of safe execution are modest, as summarized in Table 1. We believe that the overheads of safety are acceptable even on the highly RAM-, ROM-, and energy-constrained mote platforms.
- *(Section 4.)* The burden of Safe TinyOS on application developers is low. Just 0.74% of all the source lines within our benchmark TinyOS applications needed to be annotated or modified to create their Safe TinyOS counterparts.
- *(Section 5.)* Safe TinyOS did indeed help us to discover and fix interesting safety violations in existing TinyOS code. In addition, even when bugs are left unfixed, dynamic failure detection can enable significant increases in application availability.

2 Background

Safe TinyOS builds upon TinyOS 2, Deputy, and cXprop.

2.1 TinyOS

TinyOS 1 [12] and TinyOS 2 [18] are the dominant systems for programming wireless sensor network devices. A TinyOS application is an assembly of components plus a small amount of runtime support. Typically, a programmer writes a few custom components and links them with components from the TinyOS library. Components are written in nesC [9], a dialect of C with extensions for components, generic (template-like) programming, and concurrency. The nesC compiler translates an assembly of components into a monolithic C program, which is then compiled and optimized by GCC.

To conserve energy, a TinyOS application typically has a low *duty cycle*: it sleeps most of the time. Applications are interrupt-driven and follow a restrictive two-level concurrency model. Most code runs in *tasks* that are scheduled non-preemptively. Interrupts may preempt tasks (and each other), but not during *atomic* sections. Atomic sections are implemented by disabling interrupts.

TinyOS is popular for at least four reasons. First, nesC is quite similar to C—the predominant language of embed-

ded software. This heritage is important for user adoption but entails many of the problems normally associated with C code. Second, TinyOS provides a large library of ready-made components, thus saving much programmer work for common tasks. Third, the nesC compiler has a built-in race condition detector that helps developers avoid concurrency bugs. Finally, TinyOS is designed around a static resource allocation model, which helps programmers avoid hard-to-find dynamic allocation bugs. Static allocation also helps keep time and space overhead low by avoiding the need for bookkeeping.

Because TinyOS is popular, we chose to use it as the basis of our approach for developing more dependable software for sensor networks. This meant that we had to deal not only with a legacy language but also a legacy code base as described above. Fortunately, we found that we could exploit the properties of sensor network applications and TinyOS—including its concurrency and allocation models—to implement Safe TinyOS in a practical way.

2.2 Deputy

Deputy [3, 6] is a source-to-source compiler for ensuring type and memory safety for C code. It is based on the insight that the information necessary for ensuring safety, such as array bounds, is usually already present somewhere in the program. For instance, consider the declaration of a TinyOS 2 message-reception event:

```
event message_t *receive(message_t *msg,
                         void *payload,
                         uint8_t len);
```

A TinyOS programmer knows that `msg` points to a single message and that the storage pointed to by `payload` is guaranteed to be `len` bytes long. Furthermore, implementers of this event must return a pointer to a single message.

To use Deputy to get type safety, a programmer must inform the compiler of this previously implicit information using type annotations. Code compiled by Deputy relies on a mix of compile- and run-time checks to ensure that these annotations are respected, and hence that type and memory safety are respected. Using Deputy, the `receive` event is written as follows:

```
event message_t *SAFE
  receive(message_t *SAFE msg,
          void *COUNT(len) payload,
          uint8_t len);
```

`COUNT`'s argument is a C expression that specifies the length of the storage region referenced by the declared pointer. The `SAFE` annotation is shorthand for `COUNT(1)`, indicating a pointer-to-singleton that (in general) will not be used in pointer arithmetic.[1] Deputy supports a number of other annotations, for example to declare null-terminated strings and to indicate which branch of a `union` is selected. More details can be found in Deputy's manual [6].

Deputy transforms its input program to contain run-time checks, as needed, to ensure that type annotations are respected. The following example illustrates these checks. It contains two main parts: an implementation of a `receive` event, which is invoked to process message data, and a function called `dispatch` that signals a message-reception event. Below, the non-italicized code represents our input to Deputy. (As we describe later, nesC code is actually translated to C before being processed by Deputy.) Deputy processes the code and its annotations, and outputs a modified program that contains the lines shown in italics.

```
typedef struct {
  int len;
  char buffer[36];
} message_t;

event ... receive(...,
                  void *COUNT(len) payload,
                  uint8_t len) {
  int i = ...;
  if (i >= len) deputy_fail(); // INSERTED
  if (((char *)payload)[i])
    ...;
}

void dispatch(message_t *SAFE m) {
  if (m->len > 36) deputy_fail(); // INSERTED
  signal receive(m, m->buffer, m->len);
}
```

The check in the `receive` event ensures that the `payload` array access is within bounds. The second check prevents `dispatch` from passing an array that does not respect the `COUNT` annotation for `receive`.

An important characteristic of Deputy is that it does not change C's data representation. This is in contrast to CCured, for example, which may replace a regular pointer in a C program with a "fat pointer" that contains bounds information needed to ensure safety [20]. Deputy's behavior is important to our Safe TinyOS system for two reasons. First, it helps to keep memory requirements to a minimum, and it helps programmers to understand the storage requirements of their code. Second, it allows safe code (output by Deputy) to interoperate with *trusted code*. Trusted code is either existing binary code compiled without Deputy's runtime checks (e.g., a library), or snippets of C code within a file compiled by Deputy that need to violate type safety for one reason or another. For instance, TinyOS code accesses memory-mapped I/O locations by casting integers to pointers, a classic unsafe operation. Deputy allows this as long as the cast is marked appropriately: for example,

```
*((volatile uint8_t *) TC(32)) = 1;
```

`TC` marks an expression that may be freely cast to any pointer type; Deputy refers to these as *trusted casts*. Deputy supports a handful of other trust annotations as well, which are described elsewhere [6].

2.3 cXprop

cXprop [4] is our static analyzer and source-to-source optimizer for embedded C programs. Its optimizations include propagating constant scalars and pointers as well as removing useless conditionals, variables, synchronization, indirec-

[1] Strictly speaking, the `SAFE` declarations are not necessary, as Deputy assumes that unannotated pointers point to a single element.

tion, arguments, and return values. cXprop is based on an interprocedural and flow-sensitive, but path- and context-insensitive, dataflow analysis. To mitigate the effects of context insensitivity, a function inlining pass can be performed prior to analysis. This is particularly useful for TinyOS programs, which tend to contain many small functions. cXprop is built on CIL [21], a parser, type-checker, and intermediate representation for C.

cXprop's analysis tracks dataflow in the *value set* and *pointer set* abstract domains. These domains represent abstract values as explicit sets of concrete values. For example, if cXprop determines that x has the value set $\{-1, 3, 52\}$ at some program point, then in all possible executions of the system, x must take one of these values at that point. If the cardinality of a value set exceeds a predetermined value, the abstract value goes to $\perp$, the element of the domain representing the set of all possible values. Increasing the maximum size of value sets improves analysis precision but slows down cXprop. For this paper we set the maximum set size to 16, which empirically achieves a good balance between performance and precision. The pointer set domain, which serves as the basis for both must-alias and may-alias analyses, is analogous to the value set except that it contains special support for pointers about which nothing is known except that they are non-null. The maximum size for pointer sets is 64.

cXprop tracks the flow of values through scalars, pointers, structure fields, and arrays, including those with global scope. Arrays are modeled using a *collapsed* representation in which a single abstract value summarizes the contents of all array cells.

A particularly useful feature of cXprop when applied to TinyOS applications is that it can soundly analyze variables that are shared between interrupt handlers and the main program. It does this by explicitly taking into account the implicit control flow edges between the main context and interrupt handlers (and between different interrupts, if an application allows nested interrupts). The naïve approach of adding a flow edge from each node in the program graph to each interrupt handler results in prohibitive analysis times. cXprop, on the other hand, provides efficient static analysis by exploiting the insight that flow edges to interrupt handlers only need to be added at the end of each nesC atomic section.

3 Practical Safety for TinyOS Applications

Safe TinyOS is our software platform—component set and toolchain—for sensor network applications that are fail-fast with respect to software defects that cause memory access errors.

3.1 Safe TinyOS toolchain

An ordinary, unsafe TinyOS application is compiled by invoking the nesC compiler, which translates a collection of components into a monolithic C file that is then compiled by GCC. In contrast, Figure 1 shows the Safe TinyOS toolchain for producing efficient, safe programs. Safe TinyOS uses a modified nesC compiler to process annotated nesC components (Section 3.2) and then uses four new source-to-source

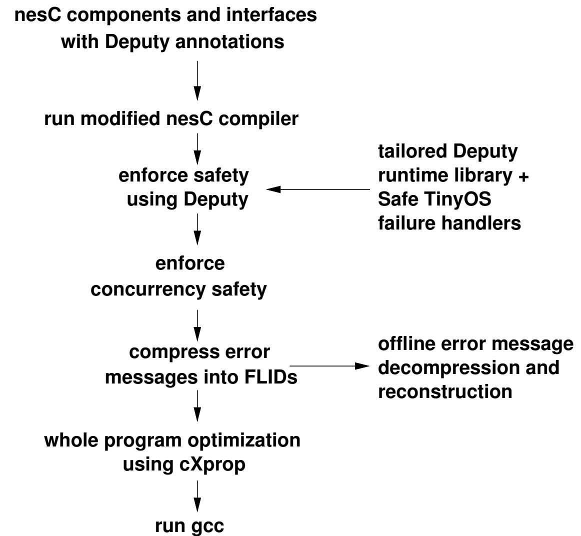

Figure 1: The Safe TinyOS toolchain

transformation steps. The first adds safety checks by running the Deputy compiler. The second ensures that type safety is maintained under interrupt-driven concurrency (Section 3.3). The third compresses bulky diagnostic information about safety violations (Section 3.4), and the fourth performs whole-program optimization using cXprop (Section 3.5).

The Safe TinyOS toolchain is easy to invoke. For example, to build a safe application for the MicaZ platform, a developer simply types "`make micaz safe`" rather than the customary TinyOS build command "`make micaz`."

3.2 Supporting Deputy annotations in nesC

In Safe TinyOS components, Deputy annotations appear in nesC source files. These source files are translated to (Deputy-annotated) C code by our modified nesC compiler. Deputy's annotations do not have any effect on nesC's compile-time checking and code generation, but they must be preserved and sometimes translated by the nesC compiler.

The contrived nesC component shown in Figure 2 demonstrates the issues involved. The translation to C involves adding a unique prefix (`Mod1$`) to all global symbols and their uses, including the use of `length` in the second `COUNT` annotation. Note however that the use of `length` in the first `COUNT` annotation is not renamed, as it refers to a field in the same structure.

Preserving the Deputy annotations turns out to be relatively straightforward: Deputy's annotations are actually C macros that expand to uses of GCC's `__attribute__` extension, which is already supported and preserved in nesC's output. However, doing the appropriate renaming is not straightforward as it requires understanding Deputy's non-standard rules for C expressions, e.g., that names such as `length` are looked up in the current structure (if any) before checking in-scope variables. Furthermore, Deputy argument expressions sometimes cause compile-time errors in the nesC compiler, e.g., by referring to unknown variables (according to

```
// nesC module 'Mod1'.
module Mod1 { }
implementation {
  // Annotation using a structure field
  struct strange {
    int length;
    void *COUNT(length + sizeof(struct strange))
          data;
  } s;

  // Annotation using a module variable
  int length;
  uint8_t *COUNT(length) buffer;
}
```

```
// C code output by our modified nesC compiler.
struct Mod1$strange {
  int length;
  void *COUNT(length + sizeof(struct Mod1$strange))
        data;
} Mod1$s;

int Mod1$length;
uint8_t *COUNT(Mod1$length) Mod1$buffer;
```

Figure 2: A Deputy-annotated nesC file and its translation to C. The nesC compiler must translate the annotations.

the standard C scoping rules).

Our current implementation is a simple, quick fix to these problems: we modify the nesC compiler to suppress compile-time errors while parsing the `__attribute__` declarations.[2] When outputting the C code for arguments to `__attribute__`, our modified nesC compiler only renames those variables that were found using C's scoping rules, and leaves the others unchanged. In practice we have found that this works well—for instance, the example in Figure 2 is handled correctly. However, if the module-level `length` variable were declared before `struct strange`, the output would be incorrect. In the near future, we will extend nesC to understand Deputy's annotations and their scoping rules.

3.3 Handling concurrency

Deputy enforces safety for sequential programs only. Concurrent code can subvert the type system by, for example, modifying a shared pointer between the time it is checked and the time it is dereferenced. To ensure safe execution, each check inserted by Deputy must execute atomically. Furthermore, as part of the atomic operation, the checked value must be copied into private temporary storage. The obvious but highly inefficient solution to this problem is to add locks around all checks inserted by Deputy.

We leveraged the nesC concurrency model to make concurrent code safe in a much more efficient way. In TinyOS applications, most variable accesses are atomic due to the nature of nesC's two-level concurrency model. Synchronous variables—those that are not accessed from interrupt handlers—are always manipulated atomically because TinyOS tasks are scheduled non-preemptively. Most asynchronous variables—those that may be accessed by an interrupt handler—are explicitly protected by atomic sections, and are therefore also manipulated atomically. Whenever a Deputy-inserted check refers only to atomically manipulated variables, that check is inherently atomic with respect to use of those variables, so no additional synchronization is required. On the other hand, when a check refers to one or more variables that are manipulated non-atomically at one or more program points (what nesC calls "racing variables"), the check requires explicit locking in order to preserve Deputy's safety invariants. Fortunately, as just described, racing variables are relatively rare by design in TinyOS code.

Thus, our technique for implementing safety efficiently is to add explicit locking only on checks that involve racing variables. The nesC compiler includes a static analysis that finds racing variables in a TinyOS application. However, nesC's analysis is unsound because it does not find racing variables that are accessed through pointers. To overcome this problem, we developed a new race condition detector that is sound in the presence of pointers. We then implemented a program transformation that adds locking to ensure that safety checks on racing variables are atomic. Our transformation also ensures that the checked value is read into a temporary variable (that is not accessed concurrently), which is used in subsequent computation. In this way, the Safe TinyOS toolchain prevents modify-after-check attacks on the type system without introducing a significant amount of run-time overhead.

3.4 Handling safety violations

When a non-embedded program tries to violate safety, the Deputy runtime prints a verbose error message to the console. There are two problems with this strategy for sensor network programs. First, message strings and other metadata are stored with the running program, where they use precious memory. Second, sensor network platforms lack an output device suitable for displaying error messages.

For Safe TinyOS, we replace Deputy's error-handling routines with a custom, resource-conserving system that works as follows. As illustrated in Figure 1, we use Deputy to produce the initial, safe C code for the applications we compile. The code that is output by Deputy contains special error-handling statements, inserted by Deputy, that use verbose strings to describe run-time safety violations. These strings identify the source locations of errors, the error types, and the `assert`-like safety conditions that fail. We wrote a separate tool, also shown in Figure 1, to extract these failure messages from the code that Deputy produces. It replaces the verbose but constant strings in our trusted error-handling code with small integers that represent those strings. We refer to these integers as *fault location identifiers*, or *FLIDs*. In addition to changing the error-handling code in our embedded applications, our tool outputs a separate file that maps FLIDs back to the complete strings that they represent. Using this file—a simple lookup table—a FLID can be turned back into a verbose failure message by a program

[2] Suppressing these errors will not cause any errors to be missed, as the attributes will be subsequently checked by GCC or Deputy.

that runs separately from the sensor network, e.g., on a developer's PC. In effect, we implemented a lossless data compression scheme to reduce the size of failure messages within Safe TinyOS applications without reducing the amount of information that those messages convey.

We make FLIDs available to developers in two ways. First, for platforms with three controllable LEDs, we disable interrupts, convert the FLID into base-4, and report it via the mote's LEDs. Second, we optionally create a network packet containing the FLID and attempt to send it over the platform's radio and also to an attached PC over the serial port. After reporting the FLID, various options are available. For debugging purposes, we program a node to display its FLID in an infinite loop. For deployments, we generally configure a node to broadcast its FLID for a short period of time, optionally write the FLID into flash memory, and then reboot or halt.

Consider the following example in which a mote encounters a safety violation while executing in the radio stack. Clearly it cannot at this point successfully use the radio to send a FLID to its base station. Instead, the mote blinks the FLID as a sequence of eight base-4 digits, which a human can provide to our command-line diagnostic tool:

```
decode_flid 00131303
```

`decode_flid` produces an error message nearly identical to the one that the Deputy runtime might have printed directly, if the mote had console I/O and other resources:

```
tos/chips/cc2420/CC2420ReceiveP.nc:241:
  CC2420ReceiveP$receiveDone_task$runTask:
    Assertion failed in upper bound coercion:
      CC2420ReceiveP$m_p_rx_buf->data + length <=
      CC2420ReceiveP$m_p_rx_buf->data + 28
```

In other words, the variable `length` is out-of-bounds with respect to a receive buffer's `data` field at line 241 of `CC2420ReceiveP.nc`.

3.5 Whole program optimization

The Deputy compiler inserts safety checks in two steps. First, it introduces many checks into an application's code. Second, it uses an aggressive optimizer to remove as many of these checks as possible, when they can be shown to be unnecessary. Even so, there is significant residual overhead.

To further reduce code size and run-time overhead, we process the code output by Deputy using cXprop. For Safe TinyOS, cXprop serves as a powerful whole-program optimizer. Unlike Deputy's optimizer, which attempts only to remove its own checks, cXprop will remove any part of a program that it can show to be dead or useless.

Precise analysis of pointers is important to Safe TinyOS because many of Deputy's checks involve pointer arithmetic. Before we tailored cXprop for Safe TinyOS, cXprop attempted to represent pointers into arrays precisely: for example, it would distinguish between the pointer values `&a[3]` and `&a[4]`. This tended to result in large pointer sets, causing cXprop to lose precision when the maximum pointer set size was exceeded. (See Section 2.3.)

To overcome this problem, we modified cXprop to interpret a pointer into an array as pointing to an arbitrary element of that array. Although it may seem counter-intuitive, weakening the precision of the pointer set domain in this fashion improved the precision of cXprop as a whole.

Change	Occ.	SLOC
TinyOS Components		
annotations:		
COUNT	17	15
SAFE	53	45
SIZE	8	8
trust annotations (TC, etc.)	18	18
code modifications (incl. annots):		
getPayload	41	62
packet access	8	19
other	4	9
TinyOS Interfaces		
annotations:		
COUNT	11	11
SAFE	34	33
Applications		
annotations:		
COUNT	1	1
SAFE	4	3
code modifications (incl. annots):		
getPayload	13	13
Total *without double-counting*	**182**	**193**

Table 2: Summary of changes for Safe TinyOS

4 The Cost of Safety

Making TinyOS applications safe incurs costs in two primary areas: source-code modifications and mote resources. This section evaluates these costs.

4.1 Code annotations and modifications

Table 2 summarizes the changes that we made to "plain" TinyOS 2 components and applications in order to create their Safe TinyOS counterparts. Although the details of the table data are somewhat subtle (as described below), the two main messages are clear. First, the great majority of our changes occur in the core parts of TinyOS—parts that are not typically changed by TinyOS application developers. Application programmers can inherit and benefit from these changes "for free." Second, as shown at the bottom of the table, the total size of all our changes is very small. We changed 193 source lines—just 0.74% of all the lines referenced in our test applications—to create Safe TinyOS.

The TinyOS source code consists of (1) the implementation of a core component set, available for use by applications, (2) the declarations that define the interfaces to those components, and (3) a set of example applications (the `apps` directory in the TinyOS source tree). For each part, Table 2 describes our changes in terms of *annotations* and *modifications*. An annotation is the insertion of a single Deputy type annotation. A modification is a change to nesC code, possibly including (but not merely being) the insertion of Deputy annotations. Thus, the two categories are overlapping—an

annotation that is part of a larger modification is counted twice—but each includes changes that the other does not.

For each kind of change, Table 2 shows the number of occurrences and the total number of source lines of code (SLOC) that were affected. An occurrence of an annotation is a single Deputy directive, whereas an occurrence of a modification is a locus of changed nesC source lines. To get a total number of occurrences and changed lines of code, one cannot simply tally the columns: some changes are counted in both the annotation and modification views, and some lines contain more than one kind of change. The totals at the bottom of Table 2 are computed so as to avoid such double-counting.

Altogether, the TinyOS 2 applications that we compiled include 253 nesC files for a total of 26,022 lines of nesC code. Thus, only 0.74% of the nesC source lines, or one in 135 lines, required a change. Most changes are extremely straightforward—many are just a single Deputy annotation. On the other hand, we had to perform two systematic changes that affected the whole TinyOS tree (the "getPayload" and "packet access" lines in Table 2).

The first of these systematic changes is an API modification. TinyOS 2 defines a `Packet` interface that contains:

```
command void *getPayload(message_t *msg,
                         uint8_t *len);
```

`getPayload` returns a pointer to the payload portion of message buffer `msg`, and sets `*len` to the number of bytes available in the payload. The "obvious" Deputy annotation for this command is:

```
command void *COUNT(*len)
  getPayload(message_t *msg, uint8_t *len);
```

Unfortunately, Deputy does not allow pointer dereferences in arguments to `COUNT`.[3] To avoid this problem, we changed the `getPayload` command to:

```
command void *COUNT(len)
  getPayload(message_t *msg, uint8_t len);
```

which is defined as returning a pointer to the payload in `msg`, as long as it is at least `len` bytes long. If it isn't long enough, a null pointer is returned instead.[4]

This modification required pervasive but small changes to TinyOS components (62 SLOC) and applications (13 SLOC). We found that the change was always straightforward and that this new API was as easy—and sometimes easier—to use as the original definition of `getPayload`. As a result of our Safe TinyOS research, the TinyOS 2 Core Working Group is incorporating this API change into a future version of TinyOS (version 2.1).

The second systematic change we made was in code used in TinyOS communication layers to access packet headers and footers. Following TEP 111 [17], the typical code to access a packet header from a message buffer `msg` is:

[3]The reason for this ultimately lies in the difficulty of performing precise alias analysis for C programs.

[4]Deputy performs run-time checks for null-pointer dereferences.

Application	SLOC	Description
Blink	3,561	Toggles the LEDs
RadioSenseToLeds	10,974	Broadcasts sensor reading and displays incoming broadcasts
Oscilloscope	11,019	Data collection
BaseStation	13,976	Serial and radio comm. bridge
MViz	21,503	Multihop collection
AntiTheft	25,773	Node theft detection

Table 3: The TinyOS 2.0.2 applications in our benchmark suite. The middle column shows lines of C code output by the nesC compiler.

```
serial_header_t* getHeader(message_t* msg) {
  return (serial_header_t *)
    (msg->data - sizeof(serial_header_t));
}
```

where `data` points to the payload above the link layer and is preceded by a byte array whose size is guaranteed to be greater than that of `serial_header_t`. Deputy does not like this code because it involves crossing a structure field boundary using pointer arithmetic. Instead, such functions must be written as:

```
serial_header_t* getHeader(message_t* msg) {
  return (serial_header_t *SAFE)
    TC(msg + offsetof(message_t, data)
           - sizeof(serial_header_t));
}
```

Note that this is logically equivalent to the original code and does not affect the function's interface.

There are 8 such functions in TinyOS components we use, leading to a total of 19 lines of code changes and accounting for three `COUNT` annotations, five `SAFE` annotations, and eight `TC` annotations.

In summary, we made very few changes to existing TinyOS 2 code in order to create Safe TinyOS, and the changes that we did make were generally simple and straightforward. Most changes are located in the core components of TinyOS, where they can be directly used by application developers. Because our approach fits well with existing TinyOS practices and code, we are receiving support from the TinyOS maintainers toward incorporating our changes into mainstream TinyOS.

4.2 Resource costs

We measured the resource costs of safety in applications that are compiled by the Safe TinyOS toolchain. Our results show that Safe TinyOS yields no RAM overhead and only modest overhead in code size (ROM) and CPU utilization.

Table 3 describes our benchmark suite, which derives from the applications that ship with the TinyOS 2 distribution. Each program was compiled for the Mica2 sensor network platform: a Mica2 mote includes an Atmel AVR 8-bit processor with 4 kB of RAM and 128 kB of flash memory. We measured the resulting executables as described below.

We do not report results for the MicaZ or TelosB platforms, but rather observe that these generally seem to be very similar to our results on the Mica2 platform.

We performed our experiments with Safe TinyOS based on TinyOS 2.0.2, and with Deputy and cXprop from CVS as of early August 2007. Safe TinyOS compiles applications using nesC's default GCC options whenever possible. For the Oscilloscope and AntiTheft applications, we use GCC's `-O` flag in conjunction with a collection of other optimization flags that, together, closely approximate the `-Os` optimization level that nesC prefers, but that avoid triggering some known GCC bugs. In all cases, Deputy's optimizer was turned to its strongest setting. We obtained code- and data-size results by inspecting the generated object files. We obtained CPU usage results using Avrora [28], a cycle-accurate simulator for networks of Mica2 motes.

4.2.1 Code size

Figure 3(a) shows the effect of various compilation strategies on the code size of Safe TinyOS applications, relative to a baseline of the original, unsafe applications as compiled by the default TinyOS 2 toolchain. Naïvely applying Deputy to TinyOS applications proved to be impractical because Deputy includes strings in its output that are used to print useful error messages when a safety violation is detected. The AVR GCC compiler places strings into RAM by default, and since the Mica2 motes have only 4 kB of RAM, the resulting executables almost always overflowed the available storage. Thus the graphs in Figure 3 do not include data for this default method of compilation.

In Figure 3(a), the leftmost bar in each cluster shows the code size when Deputy's error strings are placed in ROM using AVR-specific compiler pragmas. Code size is greatly bloated for all applications (often well beyond the 100% limit of our graphs), but since the ATmega128 processor has plentiful code space (128 kB) the applications can at least be run. We judged that an obvious alternative to bloating RAM or ROM with error strings—leaving out error messages—was unacceptable because debugging failures without the accompanying information is difficult. Note that an obvious minimal error message to report—the address of the failing check, which can be cheaply retrieved from the call stack—is not very useful for TinyOS applications due to the nesC compiler's aggressive function inlining.

The second bar of each cluster in Figure 3(a) illustrates the impact of compressing error messages as FLIDs: the average code bloat due to safety drops to 35%. The third bar in each cluster shows that whole-program optimization using cXprop reduces the average code-size cost of safety to 13%. Fortunately, if safety must be "paid for" with some resource, code memory (flash) is often the preferred currency since microcontroller platforms have much more flash than RAM. (RAM is precious because it often dominates the sleep power consumption of a sensor network node.) Unless program memory is near its limit, exchanging code memory for increased run-time dependability is the deal we would most often like to strike.

The final bar of each cluster in Figure 3(a) shows that the whole-program optimizations performed by our cXprop tool can reduce the code size of an *unsafe* TinyOS application by an average of 11%. Thus, cXprop represents a tradeoff. It can typically optimize a safe program so that its code size is acceptably close to that of the unsafe original, or it can shrink the unsafe program. One might reasonably measure the net cost of safety against the original baseline—the state of TinyOS practice—or the "new baseline" established by cXprop. In either case, cXprop succeeds at our goal of making safety a practical option in terms of code size.

4.2.2 Data size

Deputy's type system exploits bounds information already present in programs (Section 2.2), avoiding the need to allocate RAM to store array and pointer bounds. Therefore, as Figure 3(b) shows, Safe TinyOS without cXprop has no impact on RAM usage in our benchmark applications. (As described in the previous section, we omit results for the default Deputy compilation that leaves error strings in RAM, as this has prohibitive overhead for all but the smallest applications.) Figure 3(b) also shows that cXprop is able to reduce the RAM usage of safe and unsafe applications slightly by rendering some global variables unused, using techniques such as constant propagation and dead code elimination.

It was surprising to us that one of our safe, optimized applications uses a little less RAM than the corresponding unsafe, optimized application does. Upon investigation, we found that the dead code elimination (DCE) pass that runs as part of Deputy's optimizer covers a few cases that are missed by cXprop's DCE pass, permitting it to eliminate a few extra variables. These variables remain in the unsafe versions of the applications, because (obviously) we do not use Deputy to compile the unsafe applications.

4.2.3 Processor use

Measuring the CPU efficiency of a sensor network application is not entirely straightforward, because applications are reactive and motes tend to sleep a lot of the time. Our evaluation of processor use is in terms of applications' *duty cycle*: the percentage of time that the processor is awake. This metric is useful because the processor is a major consumer of energy. We measured duty cycles by creating a reasonable sensor network context for each benchmark, and then running it in Avrora for five simulated minutes.

Figure 3(c) shows the change in duty cycle across versions of our applications, again relative to the duty cycles of the original, unsafe TinyOS programs. On average, Deputy increases CPU usage by 24%, whereas the full Safe TinyOS toolchain increases CPU usage by 5.2%.

We expect that increasing processor use by 5.2% is acceptable in most cases. If the processor accounts for half of a mote's total energy consumption, the decrease in mote lifetime due to safety would be 2.6%. Nevertheless, we plan to continue working toward further optimizing Safe TinyOS applications. Consider for example that for the previous version of our toolchain [22], which was based on CCured and TinyOS 1, safe optimized applications actually had lower duty cycles, on average, than the original unsafe applications. We attribute our prior success largely to the fact that TinyOS 1 was coded much less tightly than TinyOS 2 is. The

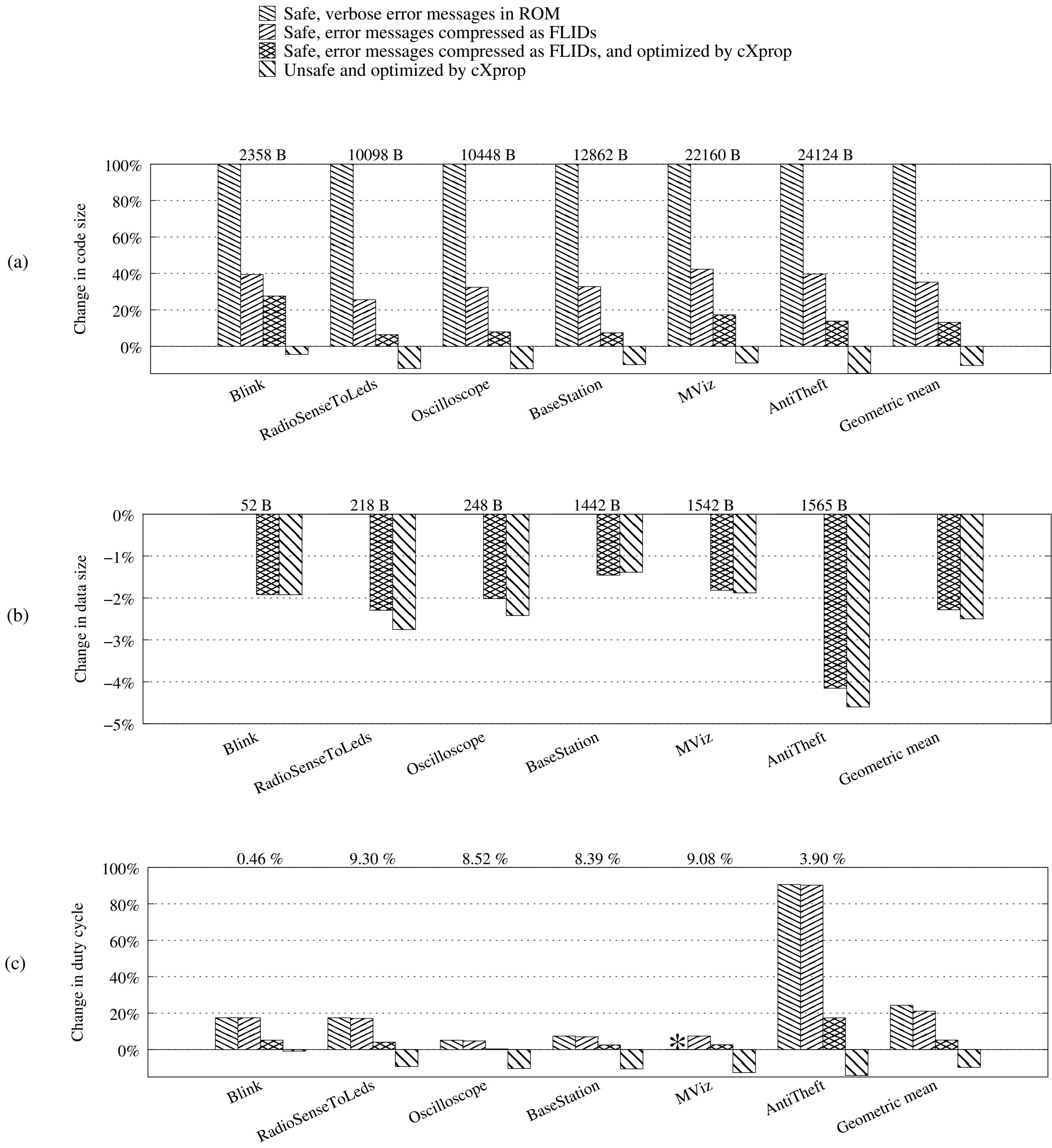

Figure 3: Resource usage relative to the unsafe TinyOS applications produced by the ordinary TinyOS toolchain. The numbers at the top of each graph show the absolute resource usage of the baseline applications. In all graphs, lower bars are better. The * indicates the application cannot be run as it does not fit in program memory.

```
message_t *rxBufPtr;

void rxData(uint8_t in) {
  cc1000_header_t *rxHeader = getHeader(rxBufPtr);
  uint8_t rxLength = rxHeader->length +
                     offsetof(message_t, data);

  // Reject invalid length packets
  if (rxLength > TOSH_DATA_LENGTH +
                 offsetof(message_t, data))
    return;
  ... code using rxBufPtr->data ...
}
```

Figure 4: Code for bug #1 (comments are ours)

extra redundancy in TinyOS 1 applications made them more amenable to optimization.

5 The Benefits of Safety

This section describes several bugs that we found in TinyOS, three of which were unknown at the time they were found. In addition, we demonstrate the utility of safe execution toward the goal of automatically increasing the availability of buggy applications.

5.1 Bug #1: ChipCon CC1000 radio stack

Context Figure 4 shows a simplified version of the `rxData` function that adds a byte to a received packet in the ChipCon CC1000 radio stack for the Mica2 platform. The `if` statement is designed to reject packets with an invalid length.

Symptoms Safe Mica2 applications would occasionally get run-time failures in the `rxData` function. The fault occurred when casting `rxBufPtr->data`, an array of `TOSH_DATA_LENGTH` bytes, to a pointer annotated with `COUNT(len)`, where `len` was equal to `rxHeader->length`. (This code is not shown in the figure.) Investigation of these failures revealed that in these cases, the packets had large, illegal lengths like `0xff`. In these cases, the addition in the assignment to `rxLength` wraps around, so those packets are not caught by the `if` statement that was intended to reject packets with invalid lengths. We note that this bug was mentioned, but not explained, in previous work [3].

Fix The TinyOS 2 maintainers moved the addition of `offsetof(message_t, data)` so that it occurs after the length test.

5.2 Bug #2: Serial stack

Context The TinyOS 2 serial stack is interrupt-driven and sends bytes out of a buffer provided by higher-level components. The module `SerialDispatcherP` implements state machines for transmission and reception of different kinds of serial packets based on a generic lower-level serial interface. The lower-level serial component signals a `nextByte` event that `SerialDispatcherP` handles by returning a new byte of data.

```
// Precondition: ‘sendIndex’ is a valid index
// into the send buffer
async event uint8_t SendBytePacket.nextByte() {
  uint8_t b;
  uint8_t indx;
  atomic {
    // This buffer access goes out of bounds
    b = sendBuffer[sendIndex];
    sendIndex++;
    indx = sendIndex;
  }
  if (indx > sendLen) {
    call SendBytePacket.completeSend();
    return 0;
  }
  else {
    return b;
  }
}
```

Figure 5: Code for bug #2 (comments are ours)

```
// Precondition: ‘current’ is a valid index into
// the queue
event void
  AMSend.sendDone[am_id_t id](message_t* msg,
                              error_t err) {
  // When the precondition is violated,
  // this access is out of bounds
  if(queue[current].msg == msg) {
    sendDone(current, msg, err);
  }
  else {
    ... print debug message ...
  }
}
```

Figure 6: Code for bug #3 (comments are ours)

Symptoms While running the safe version of the BaseStationCC2420 application on a MicaZ mote, a safety violation occurred in the `nextByte` event handler. The problem, illustrated in Figure 5, is that this function's precondition is violated because (in code not shown) a large buffer length is accidentally passed into the serial code. This leads to the `sendLen` variable being set to an incorrect value, causing subsequent buffer indexing to go out of bounds.

Fix The TinyOS 2 maintainers have incorporated an explicit check for inappropriate buffer lengths in the serial stack.

5.3 Bug #3: Unexpected event

Context The `AMQueue` component performs round-robin scheduling of the network link among multiple clients. This component maintains a state variable `current` to keep track of which client is currently being serviced. This variable is set to an out-of-bounds value to indicate a lack of clients.

Symptoms While running the safe version of the MViz application on a network of Mica2 motes, a safety violation oc-

```
// Loop to find a free buffer
for (i = 0; i < NUM_BUFFERS; i++) {
    if (m_pool[i].msg == NULL) break;
}
// Loop post-condition: either
//    (a) i indexes a free buffer
// or (b) i is an out-of-bounds index
//        pointing one entry past the end
//        of the buffer pool

// If case (b) holds, the following access
// violates memory safety
if (m_pool[i].msg == NULL) {
    // The following store corrupts RAM in
    // the unsafe application when case (b)
    // of the post-condition holds and the
    // null check (by chance) succeeds
    m_pool[i].msg = _msg;
}
```

Figure 7: Code for bug #4 (comments are ours)

curred when `AMQueue`'s `sendDone` event was called at a time when its precondition was not satisfied. The bug is not in `AMQueue` per se; rather, this module unexpectedly receives a `sendDone()` event when it has no clients. This event causes the `sendDone()` handler, shown in Figure 6, to use the invalid array index. This was a tricky bug to replicate because it only occurs after several minutes and only when a substantial number of nodes are involved.

Fix The TinyOS 2 maintainers have modified `AMQueue` to ignore spurious `sendDone()` events.

5.4 Bug #4: Incorrect search-failure check

Context By reading one of the TinyOS mailing lists, we learned of a research group that had been stalled due to a bug from June to August 2006 on the development of a time-synchronization and leader-election application for TelosB motes. We contacted the authors of this application, who sent us their source code. After processing by the nesC compiler, it was 13,800 non-blank, non-comment lines of C.

Symptoms The developers of this application experienced unpleasant symptoms: after about 20 minutes of execution, RAM on a node would become corrupted, causing the node to drop out of its network. Our safe version of this application signaled a fault after running for about 20 minutes. The problem, shown in Figure 7, was an out-of-bounds array access in the line that tests `m_pool[i].msg` for null.

We note that this bug was first discovered with our previous work, a CCured-based version of Safe TinyOS [22]. We have verified that our current Deputy-based toolchain also finds this error.

Fix Although this bug had proven almost impossible to find, it is easy to fix: the index variable `i` must be bounds-checked before being used in pointer arithmetic. The bug turns out to be in a third-party networking component, which explains why the application developers were not able to find it: they were focused on their own code. The bug is an unfortunate one because it only manifests when the send buffer overflows—an event that is expected to be quite rare.

After fixing this bug, the sensor network application appears to run indefinitely without any memory safety violations. The application's authors confirmed that this bug was responsible for the RAM corruption they were seeing. We note that by the time we found this bug, it had been fixed in the latest release of the third-party component. However, the fix dated from August 2006—two months too late to help the developers with whom we interacted.

5.5 Increasing application availability

As an alternative to helping developers find a bug, safe execution can—under certain circumstances—be used to increase application availability. For example, we changed the failure handler in the buggy time synchronization application from "Bug #4" above to reboot the sensor node immediately upon detecting a safety fault. This masks the bug by exploiting the ability of a sensor network node to restart rapidly. Since the bug in the time synchronization application occurs roughly every 20 minutes, and since the application is able to rebuild its soft state from neighboring nodes in about one minute, an average node availability of about 95% is achieved. In contrast, the long-term average node availability of the original unsafe application is zero, since nodes never recover from memory corruption—they simply drop off the network. Of course, this bug-masking strategy will not work if the ratio of the application's recovery time to its mean time to failure is too high.

6 Discussion

In doing this work, we found four ways in which Deputy fits well with TinyOS. One could even argue that Deputy is a better fit for TinyOS and nesC than it is for C code on desktop machines.

First, the fact that TinyOS does not use dynamic memory deallocation closes the loophole that Deputy does not check the safety of explicit memory deallocation (`free`).

Second, Deputy's type system exploits bound information that is already present in applications, avoiding the need to waste memory storing explicit bounds. This RAM savings is irrelevant on PC-class machines, but it is a substantial advantage on mote platforms where memory is often the limiting resource for application developers.

Third, nesC's interfaces reduce the burden of using Deputy's annotations. For example, it is only necessary to provide `receive`'s Deputy annotations once within the `Receive` interface:

```
interface Receive {
  event message_t *
    receive(message_t *msg,
            void *COUNT(len) payload,
            uint8_t len);
  command void *COUNT(len)
    getPayload(message_t *msg,
               uint8_t len);
  ...
}
```

Once this is written, all `Receive.receive` events declared in any Safe TinyOS program will inherit the annotations.

In contrast, when using Deputy with C, all function declarations must be annotated. This difference has important practical implications in terms of programmer effort. For example, Zhou et al. had to change 2.3% of all source lines in their study to apply Deputy to Linux device drivers [29]. As we reported in Section 4, however, we needed to change only 0.74% of the source lines in our TinyOS applications to make them acceptable for Safe TinyOS.

Finally, a useful side effect of Deputy is that it mitigates some unsoundness in cXprop. cXprop, like essentially all other program transformation tools for C code (including GCC), may change the behavior of a program that executes certain kinds of behavior undefined by the C language. For example, cXprop assumes that memory safety is respected: a program must not manipulate a variable through a stray pointer dereference. The guarantees provided by Deputy prevent precisely this kind of unsafe operation, and we believe that cXprop is sound with no additional assumptions about program behavior when it is applied to Deputized code.

7 Related Work

Safe TinyOS builds upon our earlier work [22] which was based on CCured [20] rather than Deputy. In addition, Condit et al. [3] present a one-paragraph summary of early experiences using Deputy with TinyOS. The present paper expands greatly on those results, presenting the full Safe TinyOS toolchain and evaluating its costs and benefits in much greater detail on a much larger ($> 3x$) code base.

We know of three ongoing efforts to bring the benefits of safe execution to sensor network applications. First, t-kernel [10] is an OS for sensor network nodes that supports untrusted native code without trusting the cross-compiler. This is accomplished by performing binary rewriting on the mote itself; this incurs about a 100% overhead, as opposed to 5.2% for Safe TinyOS. Second, Kumar et al. [23] provide memory protection in the SOS sensor network OS. This is efficient, but the SOS protection model is weaker than ours: it emulates coarse-grained hardware protection, rather than providing fine-grained memory safety. Third, Virgil [27] is a safe language for tiny embedded systems such as sensor network nodes. Like nesC/TinyOS, Virgil is designed around static resource allocation. The main distinction between Virgil and Safe TinyOS is that Virgil focuses on providing advanced object-oriented language features whereas our system maintains backwards compatibility existing TinyOS 2 nesC code.

Safe languages for constrained embedded systems have been around for a long time: Java Card [26] targets smart cards based on 8-bit microcontrollers, Esterel [2] is suited to implementing state machines on small processors, and Ada [13] was developed for embedded software programming in the 1970s. Sun SPOT [25] is a new Java-based hardware and software platform for wireless sensor applications. A Sun SPOT is not as constrained as a Mica2 mote: it has 256 kB RAM/2 MB flash, of which 80 kB/270 kB is consumed by the VM and runtime. Despite the existence of these languages and platforms, most embedded software is implemented in unsafe languages like C.

Language-based protection for C programs is an active area of research. Control-C [15] provides safety without runtime checks by relying on static analysis and language restrictions. Dhurjati et al. exploit automatic pool allocation to safely execute embedded software without requiring garbage collection [8], and to check array accesses [7]. SAL [11] and Cyclone [14] use pointer annotations similar to Deputy's to improve the safety of C code. SAL restricts itself to compile-time checking, limiting the bugs it can catch, whereas Cyclone requires significant source code changes. Simpson et al. [24] provide *segment protection* for embedded software: an emulation of coarse-grained hardware protection rather than fine-grained type safety and memory safety. Our work differs from these efforts by targeting the more severely constrained mote platform, by providing compressed error messages, by handling concurrency and direct hardware access, and by using aggressive whole-program optimization techniques to reduce overhead.

A large amount of research has been devoted to the problem of engineering safe and dependable computer-based systems [16, 19]. As far as we know, sensor networks are not yet deployed in very many safety-critical applications, and so our low-cost (in terms of developer time and machine resources) approach to increasing dependability is appropriate and practical. If—as seems likely—sensor networks become an integral part of safety-critical systems such as those governing emergency response and health care [5], then it will become appropriate to augment safe language techniques with heavier-weight software engineering methods.

8 Conclusion

We have presented *Safe TinyOS*, our software platform for improving the dependability of sensor network applications by enforcing both type and memory safety. Safety helps developers catch bugs before a sensor network is deployed, and—equally important—it prevents memory access errors from cascading into random faulty behaviors in the field.

Our contributions are as follows. We showed that aggressive whole-program analysis can minimize the overhead of safe execution. On average, as compared to unsafe, state-of-practice TinyOS applications, our benchmark Safe TinyOS applications require 13% more ROM, need 2.3% *less* RAM, and use 5.2% more CPU. These overheads are low compared to existing work that we are aware of for enforcing safety for C. We developed a technique based on sound race-condition detection for efficiently enforcing safety under interrupt-driven concurrency. We showed that the nesC language can be modified to support programmer-supplied safety annotations, and we showed that in the TinyOS applications we studied, only 0.74% of their source lines require annotations or modifications for safety. We found significant, previously unknown bugs in TinyOS. We ported Safe TinyOS to the most popular TinyOS platforms: Mica2, MicaZ, and TelosB. We developed several schemes for handling safety violations, and we support compressed error messages that have little impact on application resource us-

age and avoid sacrificing information content. Finally, we showed that safety can increase the availability of sensor network applications even when the underlying bugs remain unfixed.

Acknowledgments

Jeremy Condit, Matt Harren, Scott McPeak, and George Necula provided valuable assistance and patches for CIL and CCured. We thank Lin Gu, Jay Lepreau, Ben Titzer, and Jan Vitek for providing useful feedback on our ideas and writing. This material is based upon work supported by the National Science Foundation under Grant Nos. 0448047, 0410285, and 0615367.

References

[1] Godmar V. Back and Wilson C. Hsieh. Drawing the red line in Java. In *Proc. of the Seventh Workshop on Hot Topics in Operating Systems (HotOS)*, pages 116–121, Rio Rico, AZ, March 1999. IEEE Computer Society.

[2] Gérard Berry. The foundations of Esterel. In *Proof, language, and interaction: essays in honour of Robin Milner*, Foundations of Computing, pages 425–454. MIT Press, 2001.

[3] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *Proc. 16th European Symp. on Programming (ESOP)*, Braga, Portugal, March–April 2007.

[4] Nathan Cooprider and John Regehr. Pluggable abstract domains for analyzing embedded software. In *Proc. of the 2006 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 44–53, Ottawa, Canada, June 2006.

[5] David Culler, Deborah Estrin, and Mani Srivastava. Overview of sensor networks. *IEEE Computer*, 37(8):41–49, August 2004.

[6] The Deputy Project, 2007. `http://deputy.cs.berkeley.edu`.

[7] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proc. of the 28th Intl. Conf. on Software Engineering (ICSE)*, Shanghai, China, May 2006.

[8] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without garbage collection for embedded applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(1):73–111, February 2005.

[9] David Gay, Phil Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–11, San Diego, CA, June 2003.

[10] Lin Gu and John A. Stankovic. t-kernel: Providing reliable OS support to wireless sensor networks. In *Proc. of the 4th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, Boulder, CO, November 2006.

[11] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *Proc. of the 28th Intl. Conf. on Software Engineering (ICSE)*, Shanghai, China, May 2006.

[12] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 93–104, Cambridge, MA, November 2000.

[13] Jean D. Ichbiah. Preliminary Ada reference manual. *ACM SIGPLAN Notices*, 14(6a):1–145, June 1979.

[14] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proc. of the USENIX Annual Technical Conf.*, pages 275–288, Monterey, CA, June 2002.

[15] Sumant Kowshik, Dinakar Dhurjati, and Vikram Adve. Ensuring code safety without runtime checks for real-time control systems. In *Proc. of the Intl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Grenoble, France, October 2002.

[16] Nancy Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[17] Philip Levis. TinyOS Extension Proposal (TEP) 111: message_t, 2006. `http://www.tinyos.net/tinyos-2.x/doc/html/tep111.html`.

[18] Philip Levis, David Gay, Vlado Handziski, Jan-Hinrich Hauer, Ben Greenstein, Martin Turon, Jonathan Hui, Kevin Klues, Cory Sharp, Robert Szewczyk, Joe Polastre, Philip Buonadonna, Lama Nachman, Gilman Tolle, David Culler, and Adam Wolisz. T2: A second generation OS for embedded sensor networks. Technical Report TKN–05–007, Telecommunication Networks Group, Technische Universität Berlin, November 2005.

[19] Michael R. Lyu. *Handbook of Software Reliability Engineering*. McGraw-Hill, 1995.

[20] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3), May 2005.

[21] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. of the Intl. Conf. on Compiler Construction (CC)*, pages 213–228, Grenoble, France, April 2002.

[22] John Regehr, Nathan Cooprider, Will Archer, and Eric Eide. Efficient type and memory safety for tiny embedded systems. In *Proc. of the 3rd Workshop on Linguistic Support for Modern Operating Systems (PLOS)*, San Jose, CA, October 2006.

[23] Ram Kumar Rengaswamy, Eddie Kohler, and Mani Srivastava. Software-based memory protection in sensor nodes. In *Proc. of the 3rd Workshop on Embedded Networked Sensors (EmNets)*, Cambridge, MA, May 2006.

[24] Matthew Simpson, Bhuvan Middha, and Rajeev Barua. Segment protection for embedded systems using run-time checks. In *Proc. of the Intl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, San Francisco, CA, September 2005.

[25] Sun Microsystems. Sun SPOT system: Turning vision into reality. `http://research.sun.com/spotlight/SunSPOTSJune30.pdf`, 2005.

[26] Sun Microsystems. Java Card Specification 2.2.2, March 2006.

[27] Ben L. Titzer. Virgil: Objects on the head of a pin. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, OR, October 2006.

[28] Ben L. Titzer, Daniel Lee, and Jens Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proc. of the 4th Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, Los Angeles, CA, April 2005.

[29] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. Safedrive: Safe and recoverable extensions using language-based techniques. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation (OSDI)*, November 2006.