Memory Safety and Untrusted Extensions for TinyOS

John Regehr *Nathan Cooprider*
Will Archer *Eric Eide*

UUCS-06-007

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

June 30, 2006

Abstract

Sensor network applications should be reliable. However, TinyOS, the dominant sensor net OS, lacks basic building blocks for reliable software systems: memory protection, isolation, and safe termination. These features are typically found in general-purpose operating systems but are believed to be too expensive for tiny embedded systems with a few kilobytes of RAM. We dispel this notion and show that CCured, a safe dialect of C, can be leveraged to provide memory safety for largely unmodified TinyOS applications. We build upon safety to implement two very different environments for TinyOS applications. The first, Safe TinyOS, provides a minimal kernel for safely executing trusted applications. Safe execution traps and identifies bugs that would otherwise have silently corrupted RAM. The second environment, UTOS, implements a user-kernel boundary that supports isolation and safe termination of untrusted code. Existing TinyOS components can often be ported to UTOS with little effort. To create our environments, we substantially augmented the CCured toolchain to emit code that is safe under interrupt-driven concurrency, to reduce storage requirements by compressing error messages, to refactor direct hardware access into calls to trusted helper functions, and to make safe programs more efficient using whole-program optimization. A surprising result of our work is that a safe, optimized TinyOS program can be faster than the original unsafe, unoptimized application.

Memory Safety and Untrusted Extensions for TinyOS

John Regehr Nathan Cooprider Will Archer Eric Eide

University of Utah, School of Computing

Abstract

Sensor network applications should be reliable. However, TinyOS, the dominant sensor net OS, lacks basic building blocks for reliable software systems: memory protection, isolation, and safe termination. These features are typically found in general-purpose operating systems but are believed to be too expensive for tiny embedded systems with a few kilobytes of RAM. We dispel this notion and show that CCured, a safe dialect of C, can be leveraged to provide memory safety for largely unmodified TinyOS applications. We build upon safety to implement two very different environments for TinyOS applications. The first, Safe TinyOS, provides a minimal kernel for safely executing trusted applications. Safe execution traps and identifies bugs that would otherwise have silently corrupted RAM. The second environment, UTOS, implements a user-kernel boundary that supports isolation and safe termination of untrusted code. Existing TinyOS components can often be ported to UTOS with little effort. To create our environments, we substantially augmented the CCured toolchain to emit code that is safe under interrupt-driven concurrency, to reduce storage requirements by compressing error messages, to refactor direct hardware access into calls to trusted helper functions, and to make safe programs more efficient using whole-program optimization. A surprising result of our work is that a safe, optimized TinyOS program can be faster than the original unsafe, unoptimized application.

1 Introduction

Imagine that you have deployed hundreds or thousands of networked sensors, and are actively using them to collect data. Every so often, one of your nodes executes a software bug. The exact error is irrelevant: it could be a null pointer access, an out-of-bounds array access, or an in-bounds access to a network buffer that another subsystem is using. Now consider three scenarios.

In the first, the memory error in your application corrupts RAM on the faulting node. Since sensor network nodes are based on tiny microcontrollers that lack hardware-based memory protection, any part of memory can be corrupted. In the general case the behavior of a corrupt node is Byzantine: with sufficiently many buggy nodes over a sufficient period of time, one would expect secret keys to be revealed, sensor data to be corrupted, false routes to be advertised, and so on. It can even be difficult to distinguish between failures induced by software bugs and those caused by hardware-related problems such as weak batteries. While hardware faults can often be fixed by swapping out the defective parts, software faults persistently degrade the effectiveness of a sensor network. Time is consequently lost pointing fingers, manually rebooting nodes, and debugging code.

In the second scenario, a runtime check detects the impending memory error just before it happens and control is transferred to a fault handler. The handler could, for example, send a failure report to its base station and then reboot, or else it might go into a loop where it blinks out a failure code in octal using the mote's LEDs. Either way, you are given a concise failure code that a tool running on a PC translates into a usable error message such as:

```
Failure NULL at Buggy.nc:559: MyFunc(): Null ptr
```

In the third scenario, the impending fault is trapped, and it is recognized as occurring in untrusted, user-level code. The node is not halted or rebooted. The faulting extension is terminated while other extensions continue to operate normally. After the extension's resources have been reclaimed it can optionally be restarted.

The first scenario above is representative of the kind of problems that sensor network application developers currently face. The goal of our research is to use operating system and compiler techniques to enable the second and third scenarios. The enabling technology for our work is a "red line" [3]—a boundary between trusted and untrusted code—provided by CCured [22], a safe dialect of C. We exploited this red line to create two new ways to run TinyOS applications: Safe TinyOS and UTOS. Figure 1 contrasts the costs and benefits of these two systems with the existing TinyOS.

Safe TinyOS makes existing TinyOS applications safe in a largely backwards compatible way. Compatibility problems only manifest for applications that rely on low-level behavior, such as network reprogramming, that CCured cannot show to be safe. The small Safe TinyOS kernel includes safety checks, dynamic error handlers, helper functions for accessing hardware, and a modified

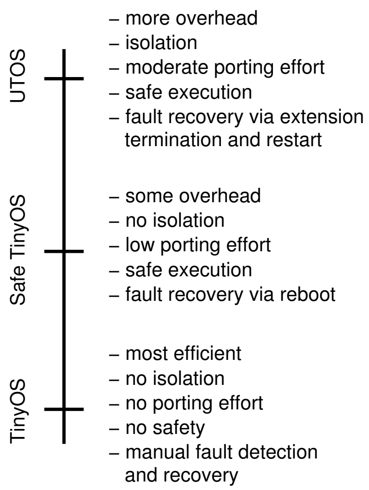

Figure 1: TinyOS as it currently exists is just one point on a spectrum from completely unsafe execution to protected, killable, user-mode execution

CCured runtime library. The rest of the TinyOS application is effectively user code. Safe TinyOS is intended to help sensor network developers by catching common kinds of memory errors. A Safe TinyOS application can subvert the kernel maliciously, e.g., by crafting a type-safety violation using assembly language. However, it is unlikely that safety will be subverted accidentally.

UTOS draws a very different red line that puts much more software into the kernel, including all interrupt handlers and device drivers. Multiple user-level extensions are supported; they communicate with the rest of the system using a narrow system call interface. An extension can be terminated synchronously, if it tries to violate the safety model, or asynchronously, if it exceeds its CPU budget. Killing and restarting an extension is about four times faster than rebooting a monolithic TinyOS application. Unlike Safe TinyOS, UTOS is intended to be airtight: a UTOS extension can execute unsafely only if our code or CCured contains a bug.

Our contributions include a number of innovations that, together, make language-based safety practical on sensor network nodes. First, code generated by the default CCured compiler is unsafe under interrupt-driven concurrency because safety checks and subsequent uses are not atomic. We developed a strategy for adding the necessary atomicity that has much less overhead than does the naïve strategy of making all checks and uses atomic. Second, we used whole-program analysis to eliminate useless computation such as redundant safety checks, with the surprising result that a safe, optimized application can end up using less CPU time than does the original unsafe, unoptimized application. Third, we developed FLIDs (fault location identifiers): compressed error messages that reduce the ROM usage of safe code without compromising the quality of error reports. Finally, we automatically refactor TinyOS code so that it accesses device registers using trusted helper functions; direct device access violates CCured's safety model.

2 Background

Our research builds directly on three existing projects: TinyOS, CCured, and cXprop.

2.1 TinyOS

TinyOS [15] is a component-based operating system for sensor network nodes. Components are written in nesC [12], a C dialect that is translated into C. TinyOS is designed around a static resource allocation model, based on the insight that dynamic allocation often introduces difficult failure modes into applications. Static allocation also helps keep time and space overhead low by avoiding the need for bookkeeping.

Many sensor network programs are constrained in terms of energy, SRAM, and flash memory. Sensor network nodes—motes—typically support up to 10 KB of RAM and up to 128 KB of flash memory. The Mica2 motes that we used for our experimental results are based on the Atmel ATmega128 8-bit processor with 4 KB of RAM and 128 KB of flash, running at 7.4 MHz. Flash memory can be written, but only slowly and in blocks; it is typically updated only when a new program is loaded.

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. The nesC compiler emits a warning when any global variable that can be touched by an interrupt handler is accessed outside of an atomic section.

2.2 CCured

CCured [22] is a source to source transformer that inputs an optionally annotated C program and emits a modified version of the program that uses dynamic checks to enforce safety. A type safe program cannot conflate types, for example treating an integer as a pointer. A memory safe program cannot access out-of-bounds storage. These properties are closely related, and CCured enforces both of them. In this paper we simply refer to "safety" to mean memory safety and type safety.

Making a C program safe without causing spurious failures or sacrificing performance is difficult in the presence of powerful language features such as type casts and pointer arithmetic. The main insight behind CCured is that in most programs, most pointers do not take advantage of the full generality of C. At compile time, CCured uses a constraint-based algorithm to conservatively infer a *kind* for each pointer in a program. Some kinds of pointers, in order of increasing generality, are:

- **Safe pointers** can be assigned and dereferenced, but not cast or manipulated with pointer arithmetic. In most cases only null checks are required for safe pointers.
- **Sequence pointers** can be manipulated using pointer arithmetic, but cannot be cast (except in a few restrictive ways). Sequence pointers require null checks and bounds checks.
- **Wild pointers** are those that cannot be statically typed. They require dynamic type checks as well as bounds and null checks.

In general, programs that end up with a large fraction of wild pointers run slowly, while programs that end up with relatively few of these incur modest runtime overhead under CCured. CCured implements sequence and wild pointers by changing them into *fat pointers* that include the necessary metadata and then rewriting the application to use these.

Although the focus of CCured is on detecting safety violations, considerable infrastructure is devoted to handling these errors once they occur. A variety of actions can be taken when the program violates CCured's safety model; the simplest is to report the error to the console and then terminate the program. An error report includes the file, line number, and function name at which the error occurred, as well as a short description of the type of error. Besides halting the program, CCured can emit a warning and keep going, put the offending thread to sleep, or ignore the error altogether.

2.3 cXprop

cXprop [7] is a dataflow analyzer for C code that we developed. It is interprocedural and tracks the flow of values through struct fields, pointers, global variables, and, to a limited extent, arrays. cXprop can act as a whole-program optimizer. It is built on top of CIL [23].

Concurrency complicates dataflow analysis by adding many implicit control flow edges to a program. cXprop exploits the TinyOS concurrency model by simulating a branch to the head of every interrupt handler at the end of every nesC `atomic` section. This permits sound analysis of variables that are manipulated atomically. cXprop then simply refuses to model variables that are manipulated non-atomically—this is typically a small minority of variables that are used in tricky ways.

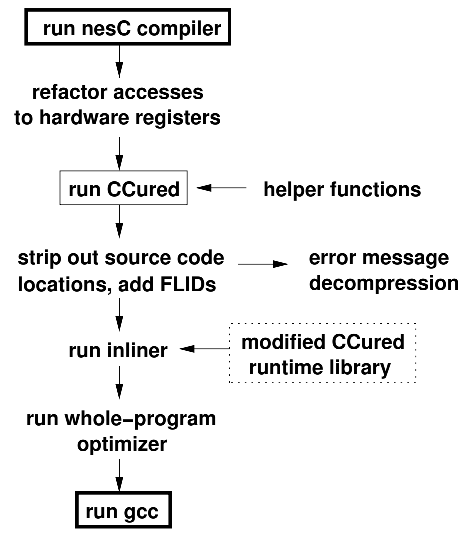

Figure 2: Our toolchain for compiling Safe TinyOS applications. Boxed tools are the ones that we did not develop. The tools in the thick boxes—the nesC and C compilers—are the original TinyOS toolchain.

3 Safety for TinyOS Applications

Figure 2 shows the toolchain the produces Safe TinyOS applications. That UTOS toolchain is largely the same but adds a few steps. This section describes the interesting parts of this toolchain.

3.1 Handling concurrency

CCured enforces type and memory safety for sequential programs. Interrupt-driven code can invalidate CCured's invariants by, for example, overwriting a pointer between the time at which it is bounds-checked and the time at which it is dereferenced. Furthermore, C programmers often expect that pointer updates occur atomically. For example, lock free data structures rely on this. CCured's fat pointers, on the other hand, cannot generally be updated atomically without explicit locking.

We modified the CCured compiler such that any time it injects code into an application, and that code references one or more global variables, the injected code is protected by a global lock. Furthermore, if the injected code is a safety check, the lock cannot be released until after the checked value has been stored in a local variable. TinyOS has inexpensive locks: on the Mica2 platform it takes just five instructions to save the state of the interrupt bit, disable interrupts, and then conditionally reenable interrupts. Even so, adding so many new atomic sections caused a 40% code space increase for some applications.

We then developed a more efficient way to enforce safety under concurrent execution, based on the observation that nearly all variable accesses in TinyOS applications are already atomic, and hence need no extra protection. The nesC compiler prints a list of variables that are accessed non-atomically. We changed the CCured compiler to read in this list and then to only insert locks around injected code that involves one or more non-atomic variables. We also needed to suppress uses of the `norace` nesC keyword, which causes the compiler to ignore potential race conditions. For the applications that we have looked at, the code size and execution time penalties of this approach are negligible.

3.2 Whole-program optimization

CCured's pointer kind inference algorithm is designed to reduce the number of dynamic checks that must be inserted into a program. Even so, a check must be inserted before every potentially unsafe operation. CCured contains an optimizer, but as we show in Section 5.1, it does not remove very many checks.

To reduce code size and runtime overhead, we post-process CCured's output using cXprop, our whole-program optimizer (Section 2). Unlike CCured's optimizer, which attempts only to remove its own checks, cXprop will remove any part of the program that it can show is dead or useless.

To support the present work we enhanced cXprop by adding a source-to-source function inlining pass. This was necessary because our context-insensitive dataflow analysis proved incapable of eliminating a significant number of CCured's checks. Inlining permits checks to be analyzed in their calling context, greatly increasing analysis precision. Section 5 evaluates the ability of our analyzer to remove CCured's checks and reports the performance and memory usage of memory-safe TinyOS programs before and after optimization.

3.3 Hardware access

The most common idiom in C for accessing a memory-mapped I/O register is to cast an integer constant into a volatile pointer and then dereference the pointer. For example, all TinyOS applications contain code of this form:

```
*(volatile uint8_t *)50U |= 0x01 << 5;
```

Direct hardware access violates safety; this kind of code must be moved below the red line. This would be trivial if we were not providing back-compatibility with existing TinyOS applications. To solve the problem we wrote a CIL extension that scans a TinyOS program looking for accesses to memory locations known to represent hardware registers, and then refactors them into calls to trusted helper functions. For example the code above is transformed as follows:

```
__cil_tmp1 = trusted_read_hw_reg_8 (50U);
trusted_write_hw_reg_8 (50U,
  ((uint8_t volatile)__cil_tmp1) | 32);
```

Since the helper functions can be inlined, they add no overhead. For example, gcc reduces both code fragments above to a single MSP430 bit-set instruction:

```
bis.b   #32,    &0x0032
```

Two kinds of code can complicate our automatic refactoring of hardware accesses. First, if the cast from integer to pointer and the dereference of the resulting pointer occur in different expressions, our syntax-driven transformation will fail. So far this problem is hypothetical; it has not happened for any code that we have seen. Second, in a few cases a non-constant integer is cast into a pointer and then dereferenced. For example, the MSP430 uses a small block of device memory to store ADC samples; it is most conveniently accessed as an array. To support this, we modified our refactoring pass to transform accesses to non-constant pointers as well as constant pointers. At runtime these non-constant pointers are bounds-checked against the start and end of device memory.

3.4 Adapting the CCured runtime library

The CCured runtime library includes a substantial amount of code: a cured application must include 2400 lines of header file code, and then it must link against a 4700-line library of support functions. There are three problems with using these header files and libraries on sensor network nodes. First, dependencies on high-level OS services such as files and signals are woven into the runtime in a fine-grained way. This code had to be manually excised since it could not even be compiled by a cross-compiler for the TelosB or Mica2 motes. Second, the runtime contains x86 dependencies. For example, several of CCured's checks cast a pointer into an unsigned 32-bit integer and then verify that it is aligned on a four-byte boundary. On the Mica2 and TelosB sensor network platforms pointers are 16 bits and do not require four-byte alignment.

The third problem with the CCured runtime is that it has RAM and ROM overhead. On a PC this overhead is negligible; on a mote it is prohibitive. When applying CCured to TinyOS programs, we tell CCured to drop all garbage collection support from the runtime. This renders CCured unsound when applied to programs that call `free`; this is no problem since TinyOS programs typically do not use dynamic allocation. Manually hacking the runtime library is effective only up to the point that all features not plausibly used by any TinyOS application are removed. Further reduction in size must be

	RAM (bytes)		ROM (bytes)	
	Mica2	TelosB	Mica2	TelosB
CCured	N/A	N/A	N/A	N/A
CCured-1	1651	318	32812	26294
CCured-1 + DCE	1347	207	3852	3266
CCured-2	71	36	23730	22936
CCured-2 + DCE	11	12	994	2076
no CCured	9	6	680	1510

Table 1: RAM and ROM usage of a minimal TinyOS application in various configurations on the Mica2 and TelosB platforms. The distributed version of the CCured runtime does not cross-compile. CCured-1 is a version of the CCured runtime that has been hacked just enough to cross-compile and CCured-2 is the version of the CCured runtime used in this paper. Versions of the runtime marked "+ DCE" add a whole-program dead-code/dead-variable elimination pass.

application-specific; for this we use a dead code elimination (DCE) pass written in CIL. Table 1 presents the storage overhead of various versions of the CCured runtime when applied to a trivial TinyOS application. The data show that both manual and automatic footprint reduction are required to achieve low overhead.

3.5 Handling dynamic errors

By default, upon encountering a safety violation, the CCured runtime displays a verbose error message and terminates the offending process. On the motes there are three problems with this. First, the information needed to produce error messages is stored within the running program, where it uses precious memory. Second, sensor network nodes lack a screen on which to display a message. Third, motes lack a standard process model.

As an alternative to verbose failures, CCured can be directed to print terse error messages, leaving out the file and line information. We consider this to be unacceptable: debugging TinyOS applications with terse error messages is almost impossible since the location of the program fault remains unknown to the programmer. To get the best of both worlds—verbose error messages with low resource usage—we wrote a tool to extract failure message information from an application and replace it with small integers that we call fault location identifiers (FLIDs). Subsequently, a FLID can be turned back into a verbose failure message by an external tool that we created. In effect, we are using an ad hoc data compression scheme to reduce the size of failure messages without reducing the amount of information that they convey.

A complication is that a given safety check inserted by CCured may not correspond to a unique error message. For example, a pointer bounds check may fail because the pointer lies above the upper bound of an array or because it lies below the lower bound. Internally, CCured uses small integers to refer to particular failure messages. When a runtime check fails, we append its failure code to the FLID associated with the check. The aggregated FLID provides sufficient information for our error conversion tool to reproduce the entire error message that the user would have seen if the sensor network node were capable of executing `printf`.

We make FLIDs available to developers in two ways. By default, we disable interrupts, convert the FLID into octal, and then report it over the mote's three LEDs. Second, we optionally also create a packet containing the FLID and attempt to send it out over the radio and also to an attached PC over the serial port. After reporting the FLID we reboot the node; another sensible option might be to put the mote into a deep sleep, saving batteries.

3.6 CCured failures

CCured cannot automatically enforce memory-safety and type-safety for all C programs. This section describes some of the problems that we encountered while applying CCured to TinyOS applications. Broadly speaking, there are three categories of problems. First, application behavior sufficiently pathological that it can be considered broken:

- The Oscilloscope application in current TinyOS 2.x CVS casts values between scalars and structs.

Second, correct application behavior that the compiler simply needs to be told to trust:

- The Nucleus network management system [33] permits remote retrieval of the contents of a node's RAM, which it accesses as a byte array.
- Network reprogramming jumps to code that arrived over the network.

Third, correct application behavior that CCured should be extended to recognize and handle:

- Several TinyOS applications use `memcpy` to copy multiple fields from one struct to another. In some cases, CCured generates a dynamic failure when the `memcpy` routine walks off of the end of the first field being copied.
- The TinyOS 2.x network stack contains code that converts a pointer from one struct member into a pointer to another member of the same struct using pointer subtraction.

Working around these problems is generally not difficult, but it does involve making changes to application code. As a last resort, CCured can be bypassed by adding trusted typecasts or entire trusted functions.

4 UTOS: Untrusted Extensions for TinyOS

Type and memory safety alone are sufficient to establish a "red line" for a simple kernel: i.e., one that enforces safety at run time. The true power of safety, however, is that it provides the needed basis for *moving* the red line. Safety makes it possible for a system designer to create new TinyOS components and draw the user/kernel boundary as he or she chooses. Due to safety, user components are unable to subvert the abstractions provided by kernel components. Safety is necessary but not sufficient for implementing the red line in software. The abstractions provided by kernel components must be carefully designed and implemented to enforce desirable properties such as isolation, resource control, and termination of user-mode code [3].

This section describes UTOS (Untrusted TinyOS), an environment for running untrusted and potentially malicious code inside a TinyOS application. Starting with a foundation of safe execution, UTOS creates a sandbox that supports termination and resource isolation for untrusted code, called *extensions*. By building on the techniques described in Section 3, UTOS ensures that an extension cannot access hardware directly, cannot use network resources inappropriately, and cannot "hijack" a node by disabling interrupts. These restrictions are in addition to the limits enforced by our implementation of safety: safety ensures that extensions interact with a node only through the well-defined UTOS interface.

Our guiding design principle was to provide extensions with an environment that is as similar as possible to the existing TinyOS programming environment. This helps programmers port existing TinyOS components to UTOS, and helped us resist the temptation to dynamically allocate resources, as SOS does [14]. We feel that this high-level design decision worked out well. Moreover, we believe that UTOS effectively demonstrates the usefulness of our approach to implementing practical, designer-chosen user/kernel boundaries in TinyOS.

4.1 Drawing a red line

The architecture of UTOS is shown in Figure 3. An *extension* is an untrusted TinyOS component, which is located in one of several *slots*. Slot numbers are used to identify loaded extensions. A set of *proxy components* provides services to extensions. Each proxy implements the interface of a standard TinyOS component, like the `Timer` component, but does so in a way that protects the kernel from the possibly harmful behaviors of extensions. The service implementations also isolate extensions from one another. We implemented the proxies by hand, and chose them according to the anticipated needs of our extensions. The proxies utilize another component, called the *backend*, that maintains resource pools for the proxies. The backend also initializes the hardware and manages the life cycle of extensions: initialization, events, and termination.

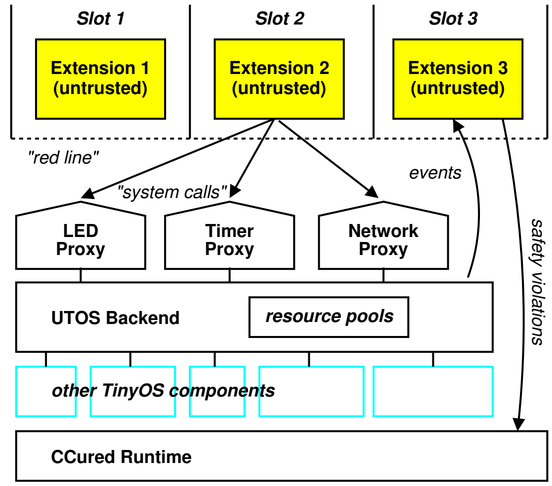

Figure 3: The architecture of UTOS

The boundary that protects the node from extensions is based on type and memory safety. These properties are enforced by our trusted compiler tools, through a combination of static analysis and compiler-inserted dynamic checks as described previously. For UTOS, we modified these tools to enforce additional restrictions:

- Extension code must not contain inline assembly language (which could be used to subvert language-based protection mechanisms).
- Extension code must not contain `atomic` sections (which disable interrupts).
- Extension code must not use CCured's "escape hatches," such as trusted type casts and trusted functions, which permit localized safety violations.
- Extension code must not refer to other components, except those that are represented in our proxy component set. References to proxied components must be rewired to refer to the corresponding proxies.
- Extension code must not refer to ordinary functions that are outside itself. This rule prevents extension code from directly accessing hardware, because the trusted hardware access functions (Section 3.3) are unavailable to extension code. This rule also prevents extension code from posting new TinyOS tasks. (The TinyOS 1.x task queue is a shared resource that a single extension could fill up.) Finally, it prevents extension code from accessing other parts of the UTOS kernel, such as the CCured runtime.

Two points should be made concerning these restrictions. First, they are completely enforced at compile time: no dynamic checking of these properties is needed. Second, the above rules apply only to "extension code," not to kernel code. Code that is inserted into an extension at compile time by our tools is trusted and therefore part of the UTOS kernel. This means, for example, that an inserted check for memory safety can disable interrupts, invoke the CCured runtime, and so on as needed.

The UTOS backend component, proxy service components, and compiler-inserted safety checks enforce the following properties at run time:

- Extensions cannot retain control of the CPU for more than a predetermined amount of time. If this limit is exceeded, control is forced back to the kernel by a timer interrupt, and the extension is terminated.
- Extension code is never invoked in interrupt mode.
- Extensions share memory only with the UTOS backend component—never with other kernel components—and only simple buffers (i.e., for network packets) are shared. The backend ensures that no buffer will be shared between two extensions.
- Extensions cannot create arbitrary network packets, which could be used to subvert or trick other nodes on the network. All packets sent by UTOS extensions are subtypes of a single packet type.
- When a compiler-inserted check detects that extension code is about to violate safety, the check invokes a special error handler. Unlike the node-rebooting handlers described in Section 3.5, the handler inserted into an extension simply terminates the extension.

UTOS does not yet support the dynamic loading of extensions, but will in the near future. (UTOS currently supports separate compilation of extensions.) The dynamic loader will ensure that any candidate extension has been properly signed by our trusted compiler.

4.2 Extensions and resources

When an extension is ready to begin, the UTOS backend invokes the extension's `init` and `start` functions. (These are part of a standard interface that is implemented by many TinyOS components.) Because extensions are currently preloaded, extensions are initialized and started when the backend component itself is initialized and started. This procedure associates the extension with events in the trusted hardware and requests any needed resources, such as timer and sensor events. Once startup is complete, an extension runs only when it receives some type of event: i.e., a timer event, or notification that a network packet has been received or sent.

During startup or subsequent invocations, an extension may invoke the proxy components shown in Figure 3. These correspond to the main I/O interfaces of a mote: LEDs, a timer service, and a radio.

The LED proxy is straightforward: an extension can freely get or set the state of the LEDs on the device. We chose not to model the LEDs as acquirable resources, so if two agents (extensions or kernel components) both use the LEDs, the output will be jumbled. One could easily redesign the LED interface to avoid this problem, if needed. Note that our design does not interfere with reporting FLIDs over a mote's LEDs: this only happens if a safety violation in occurs in the trusted kernel, at which point all extensions are implicitly terminated.

The time proxy allows an extension to acquire timer resources, which generate one-shot or repeating timeout events. Operations on timers are directed to the UTOS backend, which mediates access to the actual timing services on the mote. The backend maintains a fixed-size pool of timers and makes them available to extensions on a first-come first-served basis. We chose this policy to support flexible allocation of timers to extensions, even though it permits resource shortages among extensions. When an extension is terminated, UTOS reclaims all of the extension's timer resources.

Finally, the radio proxy allows an extension to send and receive radio network packets. The UTOS backend tracks the functions that must be called to notify extensions of packet transmission events. When an extension sends a packet, the radio proxy invokes the backend, and the backend copies the outgoing packet. This is required because the standard TinyOS interface for radio transmission is asynchronous: the `send` function returns before the packet is actually sent. Similarly, the backend component keeps a separate incoming packet buffer for each extension. When a radio packet is received for an extension, the underlying TinyOS radio components invoke the backend; the backend copies the data into the extension's incoming buffer and triggers the untrusted packet handler. This copy is needed to prevent memory sharing between an extension and the kernel radio components (which would violate the sharing property described in Section 4.1).

UTOS terminates an extension when either (1) an imminent safety violation is detected, or (2) an invocation of the extension overruns a predetermined time limit. In either case, UTOS reclaims the timer, buffer, and event table resources that are allocated to the terminated extension. Safe termination is assured by our restrictive sharing model. An interesting detail is that termination is implemented in a straightforward manner via `setjmp` and `longjmp`. UTOS is designed so that no kernel data will be lost when the stack is cleared. The `longjmp` returns control all the way back to the kernel's `main` function: `main` is the only function that is guaranteed to be on the stack whenever an extension is executing.

In our current implementation, UTOS always tries to restart extensions that have failed. We expect to implement more sophisticated strategies in the future, as part of our implementation of dynamic loading.

5 Evaluation

This section quantifies the costs and benefits of Safe TinyOS and UTOS. Although our duty cycle results are from Avrora [31, 32], a cycle-accurate simulator for networks of Mica2 motes, we also validated our Safe TinyOS and UTOS applications on real Mica2 and TelosB motes. We used a pre-release version of CCured 1.3.4, Avrora from current CVS as of February 2006, our current internal version of cXprop, and TinyOS 1.x from current CVS. All of our example TinyOS applications are from the CVS tree.

5.1 Eliminating safety checks

The CCured compiler attempts to add as few dynamic safety checks as possible, and it also optimizes the resulting code to remove redundant and unnecessary checks. In addition, our cXprop tool eliminates some checks. To measure the effectiveness of different optimizers, we transformed application source code such that for each check inserted by the CCured compiler, the address of a unique string, e.g., `__UNIQ_355`, is passed as an argument to the high-level failure handler. If an optimizer is able to prove that the failure handler is not reachable from a given check, then the string that we added becomes unreferenced and a compiler pass eliminates it from the final executable. Therefore, the number of checks remaining in a safe executable can be counted straightforwardly:

```
strings app.elf | grep __UNIQ_ | \
  sort | uniq | wc -l
```

Many CCured checks contain a number of sub-checks; in some cases cXprop eliminates only some of these. Our counting method only considers a check to be eliminated if all sub-checks are eliminated. Furthermore, function inlining causes some checks to be duplicated. Our counting method only considers a check to have been eliminated if all copies have been eliminated.

Figure 4 compares the power of four ways to optimize Safe TinyOS applications:

1. gcc, by itself;
2. the CCured optimizer, then gcc;
3. the CCured optimizer, then cXprop, then gcc; and
4. the CCured optimizer, then a function inlining pass, then cXprop, then gcc

We were surprised that gcc, on its own, eliminates so many checks. These are primarily the "easy" checks such as redundant null pointer checks. The CCured optimizer also removes easy checks and consequently it is not much more effective than gcc alone in reducing the total number of checks in executable applications. cXprop, by itself, is not amazingly effective at removing checks either. Although cXprop optimizes aggressively, it is hindered by context insensitivity: inside the analysis it merges information from all calls to a given check, such as CCured's null-pointer check, making it far less likely that the check can be found to be useless. On the other hand, inlining the checks enables context sensitivity, permitting significantly more checks to be eliminated. Our inliner attempts to avoid code bloat by respecting inlining decisions made by the nesC compiler.

5.2 Code size

Figure 5(a) shows the effect of various permutations of our toolchain on the code size of TinyOS applications, relative to the baseline: the original unsafe, unoptimized application. The first (leftmost) bar for each application shows that simply applying CCured to a TinyOS application increases its code size significantly: by around 20%–90%. The second bar is even higher, and shows the effect of moving the strings that CCured uses to construct error messages (file names, function names, etc.) from SRAM into flash memory. Unfortunately, the AVR port of gcc, which compiles applications for the Mica2 platform, places constant data such as strings into SRAM. Moving constants into flash memory by hand is error prone (the C type system does not distinguish code-space and data-space pointers) and painful (accessing code-space data requires special macros). We perform this transformation automatically with a custom CIL extension.

The third bar for each application shows the effect of using CCured's "terse" option, which suppresses all source code location information in error messages. This reduces code size but the resulting error messages are much less useful; we consider this to be an unacceptable tradeoff. The fourth bar shows the effect of compressing error messages using FLIDs. The fifth and sixth bars show that optimizing an application using cXprop, without and with an inlining pass, results in significant code size reductions. Finally, the seventh bar shows that inlining and optimizing a default, unsafe application typically reduces its code size by 10%–25%.

5.3 Data size

TinyOS applications use RAM in two ways: to store global variables and for the stack.

Static RAM consumption. CCured increases the size of a program's data and BSS segments in three ways. First, CCured makes some pointers larger by turning them into fat pointers. Second, the CCured runtime uses some memory for bookkeeping. Third, strings associated with CCured are placed into SRAM by default.

The first three bars of each group in Figure 5(b) show

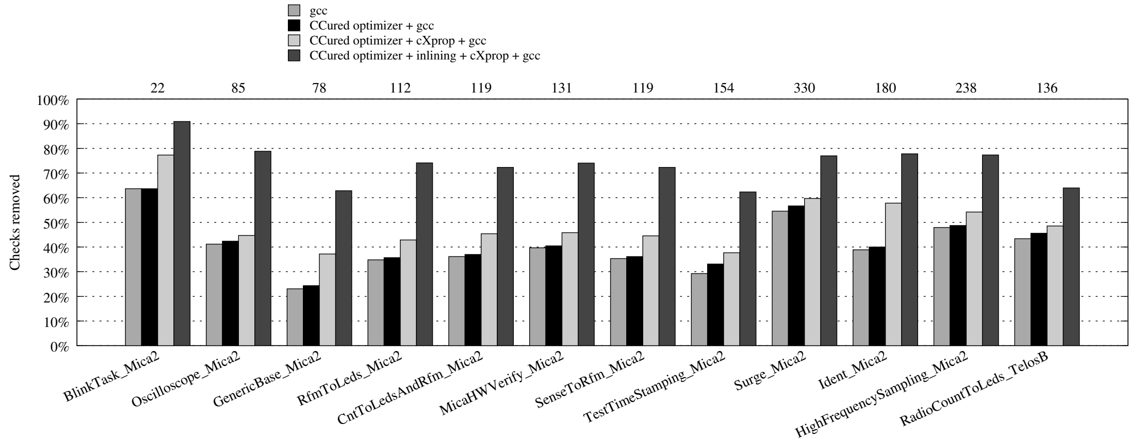

Figure 4: Percentage of checks inserted by the CCured compiler that can be eliminated using four different combinations of tools. The numbers at the top indicate the number of checks originally introduced by CCured.

that simply applying CCured to a TinyOS application results in unacceptable RAM overhead. Although we have clipped the bars at 100%, some of them are outrageously high, in the thousands of percent. The fourth bar shows that compressing error messages as FLIDs reduces RAM overhead substantially because many strings from the CCured runtime are no longer needed. These strings are actually needed by CCured's terse option, which has a few residual verbose errors in the runtime. The fifth and sixth bars show that cXprop reduces RAM overhead still more, primarily through dead-variable elimination. Finally, the rightmost bar for each application shows that cXprop reduces the amount of static data for unsafe applications slightly, by propagating constant data into code and cleaning up unused variables.

Stack RAM consumption. We were surprised to find that safe applications sometimes use a lot more stack memory than do unsafe applications, as shown in Figure 5(c). We looked into this and found that for almost all functions, the extra code introduced by CCured has little or no effect on stack memory usage. However, a few functions—typically those that perform lots of pointer manipulation in the TinyOS network stack—have many checks added to them. These checks, when inlined, add many temporary variables to the function. These added variables overload gcc's register allocator, which, instead of failing gracefully, allocates an enormous chunk of stack memory to spill everything into. Consequently, a few functions consume far more stack memory than they did previously.

The problem here is that we are performing function inlining according to decisions made by the nesC compiler. The checks added by CCured potentially invalidate these decisions by increasing function size. Although we have not done so, we believe that we could solve this problem by reevaluating all inlining decisions just before our inliner runs.

5.4 Processor use

The efficiency of a sensor network application is commonly evaluated by measuring its *duty cycle*: the percentage of time that the processor is awake. For each application, we created a reasonable sensor network context for it to run in. For example, for Surge, a multihop routing application, we simulated four Surge instances located linearly and in range of their nearest neighbors, and averaged the duty cycles. We ran each configuration in Avrora for three simulated minutes. Empirically, this is long enough for duty cycles to stabilize.

The graph in Figure 5(d) shows the change in duty cycle across different versions of our test applications. In general, CCured by itself slows down an application by a few percent, while cXprop by itself speeds an application up by 3%–10%. We were surprised to learn that using cXprop to optimize safe applications generally results in code that is about as fast as the optimized, unsafe application.

5.5 UTOS

The code size, data size, and execution time overheads of user-mode UTOS code are, by and large, not different than they are for the Safe TinyOS applications that we have evaluated up to this point. A few performance optimizations are not available to user mode code, such as using assembly language and running code in interrupt mode. Also, function inlining and cXprop optimizations are not permitted to cross the user-kernel boundary.

We implemented UTOS extensions that match the functionality provided by the TinyOS Blink, CntToLed-

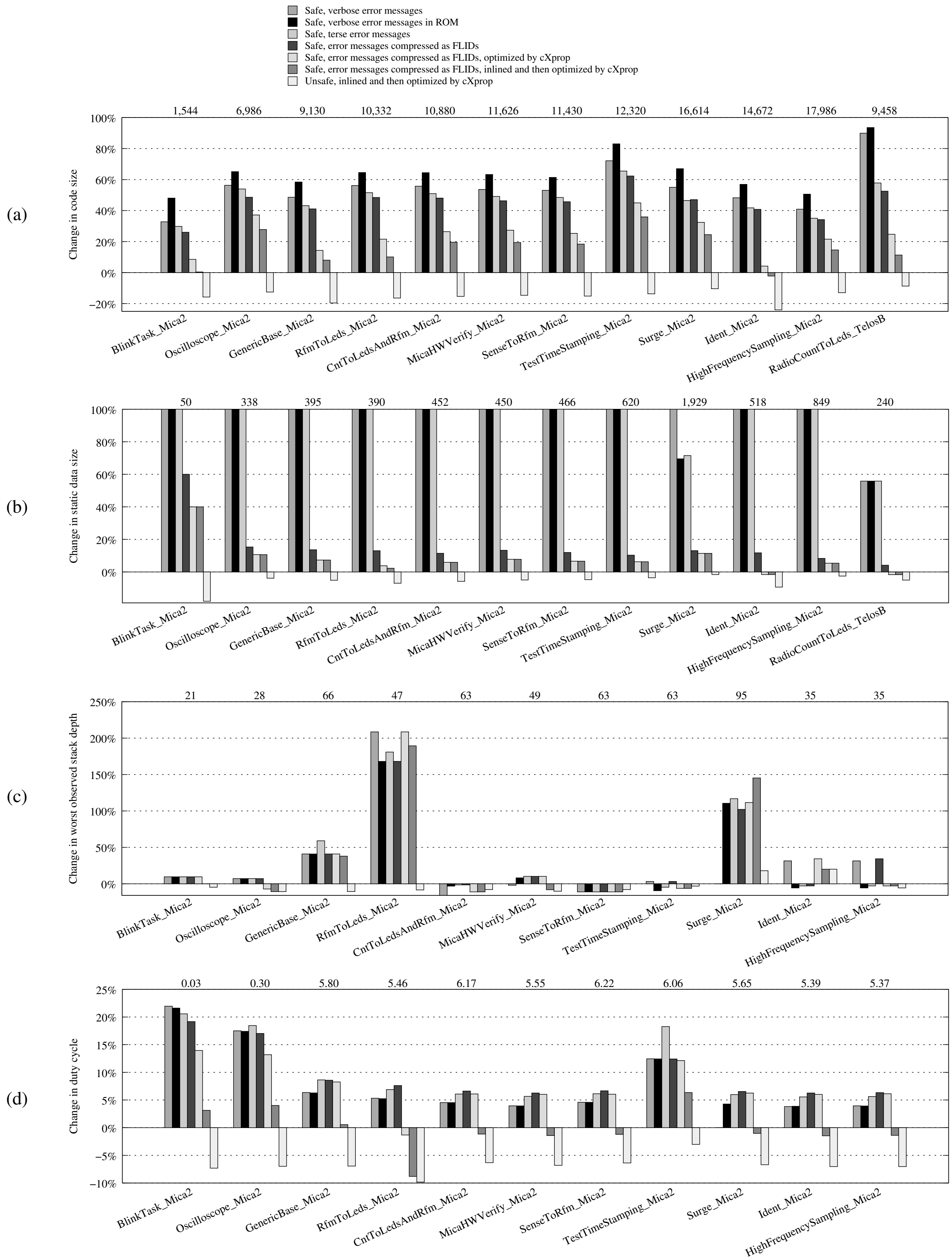

Figure 5: Resource usage relative to a baseline of the default unsafe, unoptimized TinyOS application. The numbers at the top of each graph indicate the absolute resource usage of the baseline application in bytes for memory graphs and in percent for the duty cycle graph.

Application	OS	static RAM	stack RAM	code size	duty cycle
Blink	TinyOS / UTOS	49 / 1732	21 / 107	1502 / 25174	0.0274 % / 5.95 %
CntToLedsAndRfm	TinyOS / UTOS	448 / 1793	63 / 201	10888 / 28012	6.16 % / 7.09 %
RfmToLeds	TinyOS / UTOS	390 / 1730	67 / 124	10340 / 27844	5.53 % / 6.32 %

Table 2: Resource usage for unsafe TinyOS applications and equivalent UTOS applications. All memory usage is in bytes.

sAndRfm, and RfmToLeds applications. Blink simply blinks an LED, while the other two applications exercise the send and receive sides of the network stack, respectively. Table 2 compares the RAM, ROM, and processor usage of the existing and new versions of these applications. The overheads are high because UTOS statically allocates all kernel resources that any extension might need. A larger application that fully utilizes the reserved resources would not be much larger than the simple applications that we show results for. The UTOS version of Blink has a high duty cycle because UTOS initializes the network stack without waiting for an application to do so. Here, UTOS is configured to accept up to two extensions. We have not attempted to tune the UTOS kernel for resource usage.

We did some microbenchmarking on UTOS's termination and restart capabilities. To measure the time to reboot an extension we started a timer just before the extension attempted to violate safety, and then stopped the timer after the extension had been terminated and reinitialized. The worst observed time for extension reboot is 2947 cycles, or 400 μs. The worst observed time to terminate a faulting extension without restarting it is 693 cycles, or 94 μs. In contrast, rebooting and reinitializing a Mica2 mote running the default TinyOS CntToLedsAndRfm application requires 12715 cycles, or 1725 μs. Furthermore, rebooting an entire mote wipes all volatile state, including routing tables, signal strength estimates, etc. It takes time and energy to recover this state. Rebooting a single extension leaves the state of the kernel and other modules untouched.

5.6 Finding application bugs

We did not perform any kind of systematic or thorough search for bugs in TinyOS applications. However, we did run various applications on motes and in Avrora, looking for differences in behavior between the safe and unsafe versions. While doing this we found a few bugs; we briefly describe two of them here.

The `Surge_Reliable` application in the TinyOS-1.x CVS tree contains this code:

```
ack_code[RxByteCnt +
         sizeof(ack_code) + 2 - ACK_LENGTH]
```

which, after folding constants, is equivalent to:

```
ack_code[RxByteCnt - 11]
```

The `ack_code` array has size three and upon executing this code `RxByteCnt` is in the interval [11..15]. Consequently, two elements past the array end are accessed.

The `TestEEPROM/Byte` application in the TinyOS-1.x CVS tree uses the PageEEPROM component, which has a function `sendFlashCommand` that executes a specified number of commands from a buffer in memory. The block of commands is an array of size four, but `sendFlashCommand` function is told to execute up to six commands, in which case it reads two bytes past the end of the array.

Additionally, while developing UTOS we had a bug that resulted in a `memcpy` to a null pointer. This problem interacted very poorly with the fact that the AVR architecture maps the registers into memory starting at address zero. The resulting register corruption resulted in program behaviors that were bizarre and confusing, to say the least. Turning on safety for the UTOS kernel revealed this bug right away.

6 Related Work

As far as we know, until now no safe version of C has been run on sensor network nodes, or any other embedded platform with similar resource constraints. However, other small, safe languages have been around for a long time: Java Card [29] targets smart cards based on 8-bit microcontrollers, Esterel [4] is suited to implementing state machines on small processors or directly in hardware, and Ada was developed for embedded software programming during the 1970s. Despite the existence of these languages, most embedded software is implemented in unsafe languages.

Several recent projects have focused on providing language-based protection for embedded C programs. Control-C [19] provides safety without runtime checks by relying on static analysis and language restrictions. Dhurjati et al. [9] exploit automatic pool allocation to safety execute embedded software without requiring a garbage collection. Simpson et al. [28] provide *segment protection* for embedded software, which, like SFI [34], emulates course-grained hardware protection rather than providing fine-grained type safety and memory safety. Our work differs from these efforts by targeting the smaller mote platform, by providing compressed error messages, and by handling concurrency and direct hardware access.

Although most operating systems rely on hardware-based protection, a significant amount of research has sought to implement process models through language-based protection. For example, the SPIN OS [5] is based on Modula-3, Singularity [16] is based on C#, and KaffeOS [2] is based on Java. Language-based protection has also been used inside traditional operating systems. For example, the BSD packet filter [21] is based on an interpreter for a small, safe language that runs in the kernel. STP [24] runs compiled Cyclone [17] code inside the kernel, where it enforces memory safety, time safety, and network safety. UTOS is distinguished from this previous work by targeting a platform that is orders of magnitude smaller. In TinyOS, the Maté [20] and Agilla [11] virtual machines are used to run extensions safely. UTOS differs from these by being able to run slightly modified TinyOS code in untrusted mode, as opposed to requiring applications to be rewritten in a low-level VM language. Also, UTOS extensions are fast because they are compiled to native code.

We know of three ongoing efforts to bring the benefits of safe execution to sensor network applications. First, t-kernel [13] is a sensor net OS that supports untrusted native code without trusting the cross-compiler. This is accomplished by performing binary rewriting on the mote itself. t-kernel provides safety guarantees similar to those provided UTOS, through very different mechanisms. Unlike UTOS, it sacrifices backwards compatibility with TinyOS and was reported to make code run about 100% slower. Second, Rengaswamy et al. [26] provide memory protection in the SOS sensor network OS. This is efficient, but the SOS protection model is weaker than ours: it emulates course-grained hardware protection, rather than providing fine-grained memory safety. Third, Virgil [30] is a new safe language for tiny embedded systems such as sensor network nodes. Like nesC/TinyOS, Virgil is designed around static resource allocation, and like Java Card [29] it supports objects.

A number of alternative sensor net operating systems exist, such as Contiki [10], MantisOS [6], and SOS [14]. With the exception of the memory safety work for SOS mentioned above, this work is largely complementary to ours. The developers of these systems are primarily interested in figuring out what abstractions can conveniently and efficiently support sensor network applications. Our work is based on the assumption that the existing TinyOS abstractions are good ones, and we are primarily interested in adding a user-kernel boundary that is as transparent as possible.

Besides CCured, a number of other safe dialects of C have been developed. These include Cyclone [17], Safe-C [1], Jones and Kelly's work [18], an improvement on Jones and Kelly by Dhurjati and Adve [8], and CRED [27]. This research is largely complementary to our work, which focuses on pushing safety down to very constrained embedded systems, and on exploiting language safety to run untrusted code.

7 Discussion

Why TinyOS and CCured work well together. We believe that, with our modifications and additions, CCured and TinyOS are ideally suited to one another. First, although CCured's pointer kind inference can be slow when applied to large programs, it is more than fast enough for relatively small TinyOS programs. Second, CCured's fat pointers create problems at library interfaces, but TinyOS programs do not use library functions beyond a few trivial ones like `memcpy`. Finally, CCured's reliance on a conservative garbage collector can be a problem: GC may introduce long pauses into application execution, and conservative collectors are known to leak. Because our TinyOS applications do not use dynamic memory allocation, we can drop all GC support.

Making analysis sound. We previously developed two static analysis tools for TinyOS applications: Stacktool [25], which analyzes AVR binaries, and cXprop [7], which analyzes source code. Both of these analyzers, as well as every other analyzer for C or object code that we are aware of, are unsound—that is, wrong—unless the program being analyzed conforms to a set of assumptions. For example, consider the extreme case of using static analysis to construct a callgraph for a program that contains a remotely exploitable buffer overflow vulnerability. This cannot be done. Nearly all unsoundness in our tools goes away when they are used to analyze safe code output by CCured, which cannot perform the kinds of operations that violate our analyzers' assumptions.

Using safety to help find a compiler bug. An early version of UTOS, after being run through CCured, caused the Avrora [31] simulator to throw a bounds-check exception. This happens when a program references a storage location that does not correspond to physical memory. Normally, the cause of such an error is expected to lie within the application. However, since CCured is supposed to catch all out-of-bounds memory references, we knew the application was not the problem—only an error in the CCured compiler or gcc can cause a safe application to actually crash. Inspection of the safe C code emitted by CCured showed that the translation was perfectly correct. The bug turned out to be a deeply ugly—and previously unknown—error in the AVR port of gcc that caused a call through function pointer to jump to twice the address of the intended target. Safe code limited the scope of this error to the toolchain, making it far easier to track down.

Safe TinyOS future work. Our work adds safety as a post-processing step. It might be preferable to provide safety within the nesC compiler instead, making it easier to implement new high-level safety features. For example, compiler-assisted tracking of network buffer ownership would be useful; TinyOS's buffer-swap protocol is a notorious source of bugs.

UTOS future work. Currently, UTOS supports separate compilation of extensions, but it does not support loading extensions over a network. This is one of the next features that we plan to implement. Another improvement that should be useful is to relax TinyOS's design rule that resources are allocated statically. We believe in static allocation within extensions, but probably, UTOS should dynamically allocate the resources needed by an extension when it is loaded. If extensions were compiled as position-independent code and data, then it would be possible to dynamically relocate extensions, and external fragmentation could be avoided. Finally, because battery life is a first-order concern on sensor network nodes, we would like to explore the idea of giving each extension a power or energy budget. When this budget is exceeded, the extension's activities could be throttled or it could be shut down.

8 Conclusion

We want sensor network applications to be reliable. However, existing TinyOS applications do not have access to memory protection, isolation, and safe termination: abstractions that are known to be useful building blocks for reliable software. In this paper, we have shown that these features can be supported in TinyOS, using type safety and memory safety provided by CCured as a foundation. Several innovations were required to push safe execution into tiny microcontrollers. We created a novel way to enforce safety under interrupt-driven concurrency, we compress error messages to reduce storage requirements, and we use whole-program optimization to further reduce space and time overhead.

We used our safe language infrastructure to create Safe TinyOS and UTOS, two points on a spectrum of practical TinyOS kernels. Safe TinyOS can protect a sensor network from memory errors such as array bounds violations, with the added benefits of improved error detection and diagnosis. UTOS satisfies a different purpose: it provides protection that is needed by applications that must guard some of their components (the kernel) against possible misbehavior by other components (the extensions). Terminating and restarting an extension is about four times faster than rebooting a monolithic TinyOS application. We do not claim that Safe TinyOS or UTOS represents an ideal balance between function and cost. Rather, our goal is to show that our general techniques and toolset—based on safety and powerful static analyses—are useful for implementing a *variety* of "red lines" in a cost-effective manner.

Our results show that language-based safety can be practical even for systems as small as sensor network nodes. In fact, safe, optimized TinyOS applications often use less CPU time than the original unsafe, unoptimized applications do.

Acknowledgments

Jeremy Condit, Matt Harren, and George Necula provided valuable assistance and patches for CIL and CCured. Without their help this work would have been much more difficult. We thank Lin Gu, Jay Lepreau, and Ben Titzer for providing useful early feedback on our ideas and writing. This work is supported by National Science Foundation CAREER Award CNS–0448047. This material is based upon work supported by the National Science Foundation under Grant No. 0410285.

References

[1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proc. of the ACM SIGPLAN 1994 Conf. on Programming Language Design and Implementation (PLDI)*, Orlando, FL, June 1994.

[2] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, pages 333–346, San Diego, CA, Oct. 2000. USENIX Association.

[3] G. V. Back and W. C. Hsieh. Drawing the red line in Java. In *Proc. of the Seventh Workshop on Hot Topics in Operating Systems*, pages 116–121, Rio Rico, AZ, Mar. 1999. IEEE Computer Society.

[4] G. 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.

[5] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP)*, pages 267–284, Copper Mountain, CO, Dec. 1995.

[6] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Networks and Applications*, 10(4):563–579, Aug. 2005.

[7] N. Cooprider and J. Regehr. Pluggable abstract domains for analyzing embedded software. In *Proc. of the 2006 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Ottawa, Canada, June 2006.

[8] D. Dhurjati and V. 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.

[9] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without garbage collection for embedded applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(1):73–111, Feb. 2005.

[10] A. Dunkels, B. Grönvall, and T. Voigt. Contiki—a lightweight and flexible operating system for tiny networked sensors. In *Proc. of the 1st IEEE Workshop on Embedded Networked Sensors (EmNetS)*, Tampa, Florida, Nov. 2004.

[11] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proc. of the 24th Intl. Conf. on Distributed Computing Systems (ICDCS'05)*, pages 653–662, Columbus, OH, June 2005.

[12] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. 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.

[13] L. Gu and J. A. Stankovic. t-kernel: a translative OS kernel for sensor networks. Technical Report CS-2005-09, Department of Computer Science, University of Virginia, 2005.

[14] C.-C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava. SOS: A dynamic operating system for sensor networks. In *Proc. of the 3rd Intl. Conf. on Mobile Systems, Applications, And Services (Mobisys)*, Seattle, WA, June 2005.

[15] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. 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, Nov. 2000.

[16] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, Oct. 2005.

[17] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proc. of the USENIX Annual Technical Conf.*, pages 275–288, Monterey, CA, June 2002.

[18] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proc. of the 3rd Intl. Workshop on Automated Debugging*, Linköping, Sweden, May 1997.

[19] S. Kowshik, D. Dhurjati, and V. 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, Oct. 2002.

[20] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, Oct. 2002.

[21] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. of the Winter 1993 USENIX Conf.*, pages 259–269, San Diego, CA, Jan. 1993.

[22] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3), May 2005.

[23] G. C. Necula, S. McPeak, S. P. Rahul, and W. 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, Apr. 2002.

[24] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, and T. Stack. Upgrading transport protocols using untrusted mobile code. In *Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[25] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):751–778, Nov. 2005.

[26] R. Rengaswamy, E. Kohler, and M. B. Srivastava. Software based memory protection in sensor nodes. Technical Report TR-UCLA-NESL-200603-01, Networked and Embedded Systems Laboratory, University of California, Los Angeles, Mar. 2006.

[27] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proc. of the 11th Annual Network and Distributed System Security Symp.*, pages 159–169, Feb. 2004.

[28] M. Simpson, B. Middha, and R. 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, Sept. 2005.

[29] Sun Microsystems. Java Card Specification 2.2.2, Mar. 2006.

[30] B. L. Titzer. Virgil: Objects on the head of a pin, Mar. 2006. In submission.

[31] B. L. Titzer, D. Lee, and J. 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, Apr. 2005.

[32] B. L. Titzer and J. Palsberg. Nonintrusive precision instrumentation of microcontroller software. In *Proc. of the 2005 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Chicago, IL, June 2005.

[33] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Proc. of the 2nd European Workshop on Wireless Sensor Networks (EWSN)*, Istanbul, Turkey, Jan. 2005.

[34] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. of the 14th ACM Symp. on Operating Systems Principles (SOSP)*, pages 203–216, Asheville, NC, Dec. 1993.