Static Checking for Dynamic Resource Management in Sensor Network Systems

Sensys Paper #66

Abstract

Many sensor network systems expose general interfaces to system developers for dynamically creating and manipulating resources of various kinds. While these interfaces allow programmers to accomplish common system tasks simply and efficiently, they also admit the potential for programmers to mismanage resources, for example through leaked resources or improper resource sharing. We describe a static analysis algorithm and tool that brings the safety of static resource management to systems that dynamically manage resources. Our analysis is based on the observation that programmers often use implicit *ownership* schemes to correctly manage dynamic resources. In such a scheme, each resource has a unique owner, who has both the capabilities to manipulate the resource and the responsibilities to use the resource properly and to dispose of it eventually. Our tool checks resources at compile time for violations of this ownership discipline.

We apply our tool to ensure proper management of dynamically allocated memory in programs written on top of SOS, a sensor network operating system. We have evaluated the tool on all historical versions of all user modules in the SOS CVS repository, as well as on the SOS kernel itself. Our tool generated 88 warnings of which 16 were real errors when checking user modules and 28 warnings of which 2 were real errors when checking the kernel, demonstrating the tool's utility for practical sensor network systems.

1 Introduction

Networked embedded systems, or sensor networks, are finding ubiquitous application in densely sampling phenomena — from structural properties of buildings to wildlife behavior — that were previously difficult or impossible to observe. Like embedded systems, sensor networks operate in resource and energy constrained environments with minimal operating system support. However, unlike traditional "dedicated" embedded applications, sensor network applications are reprogrammable, and applications may change in the field. For this reason, sensor network applications can benefit from operating system support for general-purpose system abstractions.

A useful class of system abstractions provides facilities to dynamically create, manage, and destroy system resources. Such abstractions can simplify application development and allow an application to naturally respond to the changing needs of its environment. Several platforms for sensor networks provide a form of dynamic resource management. For example, the SOS operating system [19] supports dynamic allocation of memory, while MANTIS OS [1] supports both dynamic memory allocation and thread creation. Frameworks built on top of sensor network systems also include components that make resources dynamically available to other parts of the system. The buffer pool used in the VanGo framework [16] for TinyOS [21] is an example of this form of dynamic resource management.

While the ability to dynamically manipulate resources increases the expressiveness of sensor network applications, this expressiveness can be a double-edged sword. Improper management of resources can lead to subtle errors that can affect both the correctness and efficiency of applications.

For example, consider the architecture of a simple sensor-network application, shown in Figure 1. Like many sensor-network applications, this application is arranged in a dataflow architecture: raw sensed data captured at the sensors moves through various filters (via the `surge` module) before being forwarded to a base station (via the `tree routing` module). In order to naturally and efficiently implement the message passing, data buffers are dynamically allocated as necessary and passed by reference rather than by copying.

If incorrectly implemented, this style of data sharing can lead to serious errors. First, a module may access a (dangling) reference to data that has been passed on and possibly freed downstream. The low-end processor hardware used in current sensor nodes does not have a memory management

unit. Thus existing sensor node operating systems for these platforms, like TinyOS [21] and SOS [19], do not support virtual memory, and a bad dereference can crash the system. Second, a module may get a data buffer from an upstream module but forget to either release it or to pass it on to the next stage for processing. Since expensive garbage-collection mechanisms are not available, such resource leaks will very soon exhaust available memory.

A key insight in these kinds of systems is that data transfer between modules typically does not lead to sharing, but instead follows a producer-consumer pattern. Therefore, sensor network programmers typically implement correct and efficient resource management using implicit *ownership-based* data access protocols. In this style, every resource has exactly one *owner* module at any point. The module that creates the resource (through some kernel call) assumes initial ownership. Ownership of a resource is explicitly transferred through kernel calls that implement message passing. Each module has the right to access the resources that it owns, but also the responsibility to either free the resource or transfer ownership to another module. A module may not manipulate resources that it does not own. If all modules obey this protocol, then there can be neither accesses to dangling pointers nor resource leaks.

While especially important for the proper management of dynamic resources, this kind of ownership model is also useful for managing static resources. For example, TinyOS lacks support for dynamic memory allocation, but statically allocated buffers are often transferred among components for efficiency reasons. TinyOS employs a split-phase approach to message passing [21]: a message pointer is passed via a call to `Send.send`, and the sender receives the `Send.sendDone` event when message transmission has completed. The calling component should not modify the message pointer in any way until receiving the `sendDone` event. Ownership is a natural protocol for guaranteeing this behavior: the calling component transfers ownership of the pointer to the callee upon a `send` call and regains ownership upon the `sendDone` event [14].

Current sensor-network systems do not help programmers to ensure that ownership protocols are properly obeyed. In most systems, these protocols are completely implicit and can be expressed only informally through programming conventions and comments. The SOS operating system includes an explicit API for expressing ownership relationships. However, programmers must take responsibility for ensuring that this API is properly used. Our experience has been that programmers often make mistakes in implementing ownership protocols, leading to critical and difficult-to-track errors.

In this paper, we present a tool for automated validation of dynamic resource management in sensor-network software. Programmers employ an explicit API along with lightweight program annotations to express ownership intentions for resources. Our tool checks each module *at compile time* for violations of the ownership protocol, providing early feedback to programmers about the potential for dynamic memory errors. The tool employs a novel suite of dataflow analyses to check the key ownership invariants: each resource has a unique owner; each resource is only manipulated by its owner; and each resource is either freed by its owner or transferred to another owner. With our tool, sensor-network programmers can make use of the expressiveness of dynamic resource management while retaining confidence in the reliability of their applications.

We have instantiated our approach as a tool for statically ensuring proper management of dynamic memory in the SOS operating system [19], which employs the C programming language. We focus on SOS memory management initially because of our experience with memory errors as users of SOS, which was our primary motivation. However, the underlying dataflow analyses are generic, and our tool is parametrized by the API for resource creation, ownership transfer, and resource deletion. Therefore, we believe that our tool can be easily adapted both to statically track ownership for other kinds of resources and to handle C-based operating systems other than SOS.

We evaluated our tool on the historic versions of all user modules in the SOS CVS repository, as well as on the SOS kernel. Overall, we ran the checker on about 46,000 lines of code. The tool identified 88 suspect memory operations of which 16 were actual memory errors in the historic versions of the user modules, as well as 28 suspect memory operations of which two were actual memory errors in the SOS kernel. The latter finding is somewhat surprising, since the SOS kernel code is relatively mature and was written by systems programming experts. In summary, our experiments illustrate the practicality of our approach for providing early feedback about memory errors in real sensor-network software.

The rest of the paper is structured as follows. Section 2 illustrates the problems for dynamic resource management via a small example and informally describes our ownership protocol that resolves these problems. Section 3 presents a precise description of this protocol and the way in which our tool enforces it. Section 4 provides our experimental results, which illustrate the utility of our tool in practice. Section 5 compares against related work, and Section 6 concludes.

2 Message Ownership in SOS

We illustrate our technique for static checking of dynamic resources through an example of dynamic memory management in SOS [19]. The SOS kernel supports dynamic linking of modules and has a dynamic memory subsystem that can be used by the modules to create and pass around messages as well as to store dynamically created state. While providing a great deal of flexibility and expressiveness to programmers, these abilities also introduce programmer obligations

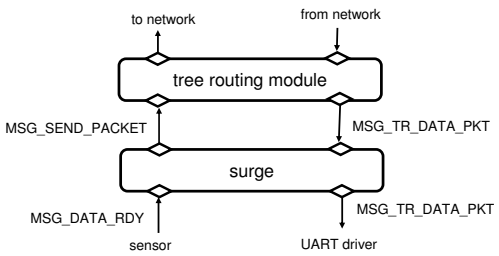

Figure 1. surge dataflow

to correctly access and manage memory. Since the underlying hardware does not have memory protection, following a dangling pointer may cause the entire system to crash. Since SOS runs on highly memory-constrained hardware, memory leaks can bring down the system very quickly.

2.1 surge

Figure 2 shows a portion of the SOS module that implements surge, a simple sensor network application that takes sensor readings and sends the readings over a multihop network to a base station [13]. The function surge_module is the entry point into surge for messages from the kernel and from other modules. The function takes two arguments: a pointer to the module's persistent state, which is saved in the kernel, and a pointer to the current message. The code for the function is a switch statement that appropriately handles each kind of message. Figure 2 shows the handlers for the messages MSG_DATA_READY and MSG_TR_DATA_PKT.

A sensor sends the message MSG_DATA_READY to the surge module when requested sensor data is ready to be read. The sensor data is passed as the data field of the message, which in general always contains a message's payload. Upon receiving this message, the surge message handler allocates a new packet (ker_malloc) to be sent to the base station and posts a message (post_long) to the tree-routing module in order to forward the sensor data. The post_long call is asynchronous, causing the kernel to package up all the given arguments into a Message structure and to schedule this message for eventual delivery.

The message MSG_TR_DATA_PKT is sent by the tree-routing module when data is received at the base station node. Upon receiving this message, the surge message handler confirms that the current node is the base station. If so, the message handler forwards the data to the UART driver via an asynchronous message send (post_net).

```
int8_t surge_module(void *state, Message *msg)
{
  surge_state_t *s = (surge_state_t*)state;

  switch (msg->type){
  case MSG_DATA_READY: // Requested sensor data ready
  {
    SurgeMsg* pkt =
      (uint8_t*)ker_malloc(sizeof(SurgeMsg));
    if (pkt == NULL) break;
    pkt->data = ... ; // ... set up message
    post_long(TREE_ROUTING_PID, SURGE_MOD_PID,
              MSG_SEND_PACKET, sizeof(SurgeMsg),
              (void*)pkt, SOS_MSG_RELEASE);
    break;
  }
  case MSG_TR_DATA_PKT:
  {
    if (ker_id() == SURGE_BASE_STATION_ADDRESS){
      uint8_t *payload = ker_msg_take_data(msg);
      post_net(SURGE_MOD_PID, SURGE_MOD_PID,
               msg->type, msg->len, payload,
               SOS_MSG_RELEASE, ker_uart_id());
      return SOS_OK;
    }
    break;
  }
  case ...: { ...; break; } // other messages
  }
  return SOS_OK;
}
```

Figure 2. SOS implementation of surge

2.2 Dynamic Memory Management in SOS

The SOS kernel provides an API for programmers to manage dynamic memory. As shown in Figure 2, the ker_malloc function acts as expected, allocating a new block of memory. The kernel also provides a ker_free function for destroying dynamically allocated memory.

In order to provide a simple form of automatic garbage collection for dynamically allocated memory, the SOS kernel imposes an *ownership* model on dynamic memory [19]. Each block of memory has a unique owner at any point in time, and the kernel maintains a mapping from each block of memory to its owner. A block's initial owner is the module that allocates that block. For example, the call to ker_malloc sets the surge module as the initial owner of the newly allocated block. When a module is removed from the system at run time, the kernel automatically frees all memory owned by that module.

The SOS kernel provides facilities for ownership transfer. Such transfer occurs in two phases. First, the owner of a block of dynamically allocated memory can explicitly *release* ownership of that block when it is passed as the payload in a message. This is accomplished by setting the SOS_MSG_RELEASE flag in the corresponding post_* call.

For example, the surge module releases ownership of the newly allocated pkt upon sending it to the tree-routing module. Second, a module can acquire ownership of a message's payload, which is stored in the data field, by calling ker_msg_take_data on an incoming message. The function returns a pointer to the message's payload. For example, if the current node is the base station, the surge module explicitly takes ownership of the given message's data under the name payload.

There are four release/take scenarios to consider. If data is both released by its sender and taken by its receiver, then ownership of the data is transferred from the sender to the receiver. If data is released by its sender but not taken by its receiver, then the kernel automatically frees the memory after the receiver's message handler completes. If data is not released by its sender but is taken by its receiver, then the sender keeps ownership of the original message and the receiver gains ownership of a new block of memory containing a copy of that data. Finally, if the data is not released by the sender and not claimed by the receiver, then the sender keeps ownership of the original message and the receiver has direct access to "borrowed" data for a limited period of time. This last case is not generally used in SOS due to the synchronization complications that can result.

2.3 Static Ownership Checking

SOS's original concept of ownership as described above provides a simple form of garbage collection at run time. While this can help reduce the complexity of managing dynamic memory, it is not sufficient to prevent memory errors such as dangling pointers and memory leaks. For example, nothing prevents a module from freeing some memory while another module (or even the same module) still has a pointer to that memory. If that pointer is ever accessed later, an invalid dereference will result. Further, garbage collection introduces the potential for more dangling pointer errors, since the removal of a module implicitly frees the memory it owns, even if other modules have pointers to that memory.

In this work, we augment SOS's ownership directives to provide a protocol governing memory management that is sufficient to ensure the absence of memory errors. Our protocol makes explicit a common programming idiom in sensor-network systems, whereby data is rarely shared but instead follows a producer/consumer model. We have built a tool that checks for violations of this protocol on each SOS module at compile time.

Informally, the rules of the protocol can be stated as follows:

- A module may only manipulate the memory that it owns.
- A module that takes ownership of a block of memory (either through ker_malloc or ker_msg_take_data) must either free that memory, release it, or store it in the module's persistent state.
- A module may only free or release memory that it owns. After a module frees or releases memory, it may not access or update that memory.

Because only the owner can manipulate or free its memory, dangling pointers are avoided. Because all memory must be either freed, released or persistently stored by its owner, memory leaks are avoided.

2.4 surge Revisited

Our tool is able to validate at compile time that the surge module properly obeys the ownership protocol. The MSG_DATA_RDY message handler allocates pkt and takes ownership. This pointer is then dereferenced in order to provide the sensor data to be sent up the routing tree. This pointer manipulation is safe since the module has ownership. The module then releases ownership by posting pkt to the tree routing module using the SOS_MSG_RELEASE tag. After this release, the module does not access pkt again and does not store it, ensuring that access to the pointer is indeed released.

The handler for MSG_TR_DATA_PKT also conforms to the protocol. When the current node is the base station, the handler explicitly acquires ownership of the message's data using ker_msg_take_data. This allows the module to manipulate the data and to pass it to the UART. The post_net call explicitly releases the data, fulfilling the module's obligation to that data. After the release, the data is no longer accessed or stored.

While the surge code is correct, small changes to the code can easily cause problems to occur at run time, and our static checker catches these potential errors. For example, suppose the handler for MSG_DATA_READY did not release ownership of pkt by setting the SOS_MSG_RELEASE flag in the call to post_long. In that case, the module would leak the memory allocated for pkt. Indeed, our checker flags this modified version of the code as erroneous, since surge would not be freeing, releasing, or storing the data for which it has taken ownership.

2.5 Function Attributes

Our analysis is *modular*: each function in a module is analyzed in isolation. To make checking of a function body precise in the presence of calls to other functions, we employ *ownership attributes* for function headers that capture the memory-related behavior of a called function. We add two attributes to the SOS API: sos_claim and sos_release. A formal parameter or return value that has the sos_claim attribute indicates that the caller must take ownership of the associated memory after a call. This annotation, for example, would be used to annotate a function that wraps a call to ker_malloc, allowing that function's callers to be properly checked without access to the function's implementation. Similarly, an sos_release attribute on a formal parameter indicates that ownership of the parameter is transferred from

the caller of the function to the callee. If a parameter does not have an ownership attribute, memory ownership is unchanged. Our tool ensures that these attributes are employed wherever necessary, when checking the implementation of each function. In practice, we have found that a small set of annotations is sufficient for precise analysis.

2.6 Generality

Memory errors in SOS were the original motivation for our work. Further, the existing ownership directives for dynamic memory in SOS facilitated the creation of our tool. However, we stress that the underlying protocol that we enforce is completely independent of both SOS and of memory in particular. For example, it would be straightforward to apply our tool to track resources other than memory in SOS, and we could similarly port our tool to enforce an ownership protocol on nesC's static buffers rather than SOS's dynamic memory.

3 Analysis Specification and Implementation

3.1 Specification

We now formalize the logical specification of our ownership-based resource management protocol. For simplicity, we specify the protocol on an idealized imperative language that includes the fundamental operations for resource management. The language's atomic statements are as follows, where `x` and `y` range over *resource handle names*:

$$y := x \mid \text{alloc } x \mid \text{free } x \mid \text{get } x \mid \text{store } x$$

Programs are constructed by sequencing statements of the above form, and we assume the presence of standard control-flow constructs including conditionals and loops.

Our language's statements abstractly capture resource-related program operations and have the following informal semantics. An assignment $y := x$ assigns the resource referenced by handle x to the handle y. The operation $\text{alloc } x$ allocates a new resource and returns a handle x to the resource. The operation $\text{free } x$ frees the resource referenced by the handle x. In addition, we assume the presence of a persistent store that holds a single resource. The operation $\text{store } x$ puts the resource referenced by handle x into the store, and the operation $\text{get } x$ assigns the resource contained in the persistent store to x.

Given this core language, we formally specify rules for proper resource usage as invariants on each dynamic program execution. A natural formalism for these invariants is *linear temporal logic* (LTL) [11, 22], which we briefly review. An LTL formula is constructed from atomic predicates using boolean operations and the *temporal operators* $\Box$, $\Diamond$, and $\bigcirc$. An LTL formula is evaluated at a given execution state with respect to a fixed execution path through the program. The formula $\Box f$ holds at state s if the formula f holds on s and each subsequent state in the path. The formula $\Diamond f$ holds at state s if the formula f holds on s or some subsequent state in the path. The formula $\bigcirc f$ holds at state s if the formula f holds on the successor state of s along the path.

For a handle x, we write $\mathsf{access}(x)$ for any operation that syntactically accesses the resource referenced by x, that is, one of $\text{free } x$, $\text{store } x$, or an assignment with x on the right-hand side. We also introduce a relation $\mathsf{alias}(x, y)$ on pairs of handles: $\mathsf{alias}(x, y)$ holds at an execution state iff handles x and y refer to the same resource in that state. We now define two key correctness invariants for proper resource management.

No Leaks. The first property formalizes the absence of resource leaks by enforcing that along every program execution path, every allocation $\text{alloc } x$ is necessarily followed by either a free or a store of a handle that refers to the same resource as x:

$$\Box\left((\text{alloc } x) \rightarrow \Diamond\left((\text{free } y \vee \text{store } y) \wedge \mathsf{alias}(x, y)\right)\right)$$

No Dangling Pointers. The second property states that once a resource has been freed, it is never accessed again:

$$\Box(\text{free } x \rightarrow \bigcirc(\Box(\neg(\mathsf{access}(y) \wedge \mathsf{alias}(x, y)))))$$

3.2 Implementation

Our tool checks SOS programs at compile time for violations of the above two properties. The tool performs a static dataflow analysis for each property on a program's control-flow graph (CFG), which statically represents all possible execution paths. The alloc and free operations are respectively represented by `ker_malloc` and `ker_free`. To identify a module's persistent state, we provide a new attribute `sos_state` for formal parameters. For example, the `state` argument to a module's event handler would be annotated with this attribute.

Our dataflow analyses are implemented in the CIL front end for C [23], which parses C code into a simple intermediate format and provides a framework for performing analyses on the intermediate code. Each analysis is *modular*, considering each module independently and analyzing the CFG for each procedure within the module in isolation. Care must be taken for our static checker to conservatively approximate the dynamic conditions that must be satisfied. Two notable issues are the treatment of pointer aliasing and of procedure calls, which we discuss in turn.

First, the invariants described above depend on dynamic alias information, which cannot be exactly computed at compile time. Instead, two standard approximations are *must-alias* analysis and *may-alias* analysis. A must-alias analysis underapproximates the dynamic alias relations: if a must-alias analysis determines that x and y alias at a particular program point, then $\mathsf{alias}(x, y)$ definitely holds at any runtime execution state corresponding to that program point. A may-alias analysis overapproximates the dynamic alias relations: if a may-alias analysis determines that x and y *cannot*

alias at a particular program point, then $\mathsf{alias}(x, y)$ definitely does not hold at any run-time execution state corresponding to that program point.

Our checker requires both kinds of static alias approximations. In the first property described above, alias information is used to ensure that something definitely happens, namely that an allocated resource is eventually freed. Therefore, in this case we approximate the true alias information with must-alias information. In the second property described above, alias information is used to ensure that something definitely does not happen, namely an access to a freed resource. Therefore, in this case we approximate the true alias information with may-alias information.

Our implementation currently uses a simple flow-sensitive must-alias analysis. For the may-alias analysis, we use a fast flow-insensitive alias analysis provided by the CIL framework. Static alias analysis can be quite imprecise in the presence of complex pointer manipulations and pointer structures. For example, CIL's may-alias analysis does not distinguish among the fields of a structure, instead considering them to always potentially alias one another. However, our experimental results indicate that these limitations do not result in an inordinate number of false positives for the checker. This positive result is likely due to the stylized ways in which dynamic memory is manipulated in sensor-network applications, which our ownership model captures well.

Second, our simple core language used to specify the resource invariants does not contain procedure calls, which must be properly handled in our implementation. As mentioned above, our analyses are *intraprocedural*, meaning that each procedure is checked in isolation. To make such checking both correct and precise, we rely on the ownership attributes `sos_claim` and `sos_release`, as described in the previous section.

Given such attributes, the proper handling of procedure calls becomes straightforward. A procedure call statement is treated logically by the checker as an assignment from actuals to formals, followed by an assignment from the return value of the call to the left-hand-side variable (if any). A formal parameter annotated with the `sos_claim` attribute is treated as an allocation site, just as is `ker_malloc`. A formal parameter annotated with the `sos_release` attribute is treated as a disposal site, just as is `ker_free`.

The ability to perform checking modularly allows SOS application writers to obtain early feedback about the correctness of their resource management, without requiring access to the rest of the system. This is particularly important in a system like SOS, in which modules can be linked and unlinked dynamically. In such a setting, the "rest" of the system is a moving target, so it is not really possible to consider whole-program analysis.

3.3 Limitations

As we demonstrate in the next section, our checker is useful for detecting violations of the ownership protocol on real sensor-network code. However, the checker is not guaranteed to find all such violations. Said another way, the checker can be used for finding memory errors but not for guaranteeing the absence of all memory errors. We discuss three limitations of the checker in this regard.

First, the checker does not precisely handle all of the unsafe features of the C programming language. For example, pointer arithmetic is not statically analyzed. Instead, an expression of the form $p + i$, where p is a pointer and i is an integer, is simply treated as if it refers to the same block of memory as p. If $p + i$ in fact overflows to another block of memory at run time, the checker's assumption can cause it to miss errors. These kinds of limitations are standard for C-based program analyses.

Second, there is a design choice about how to treat messages that a handler does not explicitly acquire through `ker_msg_take_data`. Technically the ownership protocol disallows these messages from being accessed at all, since the receiving module is not the owner. However, enforcing this requirement can cause spurious errors for patterns of data sharing that are in fact safe, for example when a module temporarily "borrows" data from another module within a bounded scope. Therefore, our checker does not currently enforce the requirement that a module only access memory that it owns, reducing the number of false positives but also potentially missing errors. The checker still ensures that a message handler does not access memory that has earlier been freed or released.

Finally, our modular checker does not consider the overall order in which messages will be received. However, there may be application-level protocols that determine how events are generated in the system, and these protocols can affect the correctness of ownership tracking. For example, imagine a module that responds to three events: create causes the module to allocate a block b and store it into the module's persistent store, access causes the module to access block b via the store, and delete causes the module to deallocate block b via the store. The module properly manages block b as long as the temporal sequence of events follows the regular expression $\mathsf{alloc}\ \mathsf{access}^*\ \mathsf{delete}$. This ordering ensures that the system never accesses the block before it is allocated and always eventually deletes the block.

Our tool currently only tracks data locally within each event handler. When performing this tracking, the tool assumes on entry that all data in the persistent store is owned by the module. The tool similarly considers a resource to be properly released when it is placed in the persistent store. These assumptions can miss errors, for example if an access event ever occurs before the alloc event. However, our assumptions provide a practical point in the design space that allows each event handler to be usefully checked for local violations of the ownership protocol. It will be interesting to extend our system with additional interface annotations about application-level protocols in the future [3, 20].

Table 1. Warnings in SOS user modules

Actual memory leaks	16
Missing annotations	153
Free within a loop	66
False positives	72

4 Evaluation

4.1 Quantifying Dynamic Memory Usage in SOS

First we performed a study to understand the prevalence of dynamic memory operations in the SOS source. The SOS API includes a number of functions that manipulate memory. These functions include `ker_malloc` and `ker_free` to allocate and free data, `ker_msg_take_data` and `SOS_MSG_RELEASE` used to transfer data ownership, and a handful of functions that employ these primitives to allocate specific kinds of data structures. Examining the SOS CVS head from April 2006 reveals that nearly one memory operation appears per 100 source lines of code (SLOC). Of the 46371 SLOC examined, 12990 are from modules written by end users to build applications, rather than from the implementation of SOS itself and associated drivers. The frequency of memory operations in end-user modules is even higher than the overall average, with one such operation for every 60 SLOC. These findings indicate that memory management is an important part of sensor-network programming in SOS.

4.2 Validating SOS End-User Modules

Next we ran our tool on all SOS end-user modules from the April head of the SOS CVS repository. The goal of this experiment was to demonstrate that the tool is practical for use as part of the standard development cycle and to demonstrate its ability to locate errors in real sensor-network software.

We integrated our checker in the normal SOS build process by adding a target to the Makefile that invokes the checker on the given code during compilation. The average time to check a file is less than a quarter of a second. Output from the analysis takes the form of warnings similar to those generated during other stages of compilation. The analysis generates three types of warnings:

Dangling pointer: Source code is accessing data that was released or freed at an earlier point.

Access to dead data freed in a loop: Special case of the prior warning. Limitations of CIL's may-alias analysis result in false positives when data is freed in a loop. Separating this case out alerts the user to a probable false positive. Users may also set a flag to suppress these warnings.

Memory leak: Source code neither stores, releases, nor frees data that was taken or allocated.

We applied the checker to every historic version of each user module included in the SOS CVS repository. Of the 203 historic versions of the 37 modules available, 77 versions resulted in warnings from the checker, eight versions caused the CIL parser to crash before checking could occur, and the remaining 118 versions passed the checker with no warnings. We have not yet had the opportunity to resolve the problem with CIL's parser.

A total of 307 warnings were generated by the checker during this experiment. Each warning was examined by hand and classified as an actual error or a false positive; the results are presented in Table 1. Sixteen of the warnings turned out to be real memory leaks in the code, resulting from four distinct errors that remained in place across multiple CVS versions. An example memory leak from the code is described at the end of this section.

The 291 remaining warnings were further classified to better understand the sources of imprecision in the current implementation of the checker.

Missing annotations. We ran the checker with 20 ownership attributes on functions from the core SOS API, as well as an `sos_state` attribute for the `state` argument to each module's message handler. While testing the tool on historic versions, 153 warnings were due to missing annotations. These warnings consist of 60 due to changes in the behavior of SOS API functions over time, 49 due to (currently deprecated) functions in the core SOS API that required annotations and are used in historic versions of modules, and 44 warnings from the need for additional `sos_state` attributes (for example, when a message handler's `state` argument is passed as a parameter to a helper function). All of these warnings are eliminated once the appropriate annotations are added to the code.

Free within a loop. As mentioned earlier, imprecisions in the may-alias analysis cause warnings to be signaled when data is freed within a loop. There were 66 such false positives across all historical versions of the end-user modules. As mentioned earlier, our tool includes an option to suppress these warnings, which is enabled by default.

Other false positives. After elimination of the above two classes of warnings (either by annotation or by suppression), we are left with 72 false positives, which come from three main sources. The primary culprit is CIL's flow-insensitive field-insensitive may-alias analysis, which is particularly imprecise for reasoning about deep data structures. We plan to experiment with more sophisticated alias analyses to eliminate many of these false positives.

A second source of false positives arises from functions that conditionally release data in SOS. Some functions take as input a dynamically allocated buffer and return a status code indicating whether the data was released. The caller of the function is responsible for checking this return value and, based on its value, properly treating the data. Since our analysis is not path-sensitive, these kinds of functions result in false positives. However, the presence of these warnings

Table 2. Warnings in the SOS kernel

Actual memory leaks	1
Actual dangling pointer errors	1
Missing annotations	10
Free within a loop	5
False positives	26

has led us to conversations with the SOS developers about transitioning away from this style of API.

A third source of false positives arises from application-level protocols that ensure memory is used correctly across multiple invocations of the module's message handler, but which cannot be validated modularly.

4.3 Validating the SOS Kernel

We ran a similar experiment on the SOS kernel. We ran the checker on the current version of all code required to build the core SOS kernel with a simple "blink" application for the Mica2 hardware. This configuration consists of approximately 15000 SLOC and required analyzing 37 source files, of which 16 generated warnings. A detailed listing of these warnings is provided in table 2. A total of 43 warnings are reported by the checker.

Surprisingly, although the SOS kernel code is stable and heavily exercised, the checker found two actual errors. The first is a memory leak similar to the example shown in the next subsection. Listed here is the other error, an improper access to data that has been released to another module.

```
if( post_long(fst->requester, KER_FETCHER_PID,
              MSG_FETCHER_DONE, sizeof(fetcher_state_t),
              fst, SOS_MSG_RELEASE) != SOS_OK ) {
    ...
    return;
}
cam = ker_cam_lookup( fst->map.key );
...
```

In this segment of code the variable `fst` is released by the call to `post_long` and subsequently accessed when the `if` clause fails. Both of the kernel errors found by the checker have been reported to the SOS development team. The presence of memory errors in mature and well-tested code, written by systems programmers, demonstrates the utility of an automatic tool for validating memory management.

4.4 A Memory Leak in SOS

The following is an example of a memory leak found by the checker. The error was found during testing of historic versions of the file `loader.c`, which is used in SOS to dynamically load other modules onto running nodes. In mid-October 2005 the block of code shown in Figure 4.4 was checked into CVS as part of `loader.c`. All paths through this block of code leak the `mod_op` pointer, which the module acquires ownership of through `ker_msg_take_data`. This code results in the following warning from the checker:

```
Warning: Alloced data from instruction #line 125 "loader.c"
mod_op = (sos_module_op_t*) ker_msg_take_data(msg);
if(mod_op == NULL) return -ENOMEM;
if(mod_op->op == MODULE_OP_INSMOD) {
    existing_module = ker_get_module(mod_op->mod_id);
    if(existing_module != NULL) {
        uint8_t ver = sos_read_header_byte(
                existing_module->header,
                offsetof(mod_header_t, version));
            if (ver < mod_op->version) {
                ker_unload_module(existing_module->pid,
                        sos_read_header_byte(
                        existing_module->header,
                        offsetof(mod_header_t, version)));
            } else {
                return SOS_OK;
            }
        }
    ret = fetcher_request(KER_DFT_LOADER_PID,
            mod_op->mod_id,
            mod_op->version,
            entohs(mod_op->size),
            msg->saddr);
    s->pend = mod_op;
    ker_led(LED_RED_TOGGLE);
    return SOS_OK;
}
return SOS_OK;
```

Figure 3. A memory leak in an SOS module.

```
    mod_op = (sos_module_op_t *) ker_msg_take_data(msg);
is not stored
```

After three additional revisions that do not significantly modify or fix the memory leak, a fourth revision was made in mid-December 2005. This revision expands the functionality of `loader.c` and breaks the code up into smaller functions:

```
sos_module_op_t *mod_op;
if (msg->saddr == ker_id() || s->pend) {
    return SOS_OK;
}
mod_op = (sos_module_op_t*) ker_msg_take_data(msg);
if(mod_op == NULL) return -ENOMEM;
switch(mod_op->op){
case MODULE_OP_INSMOD:
    return module_op_insmod(s,msg,mod_op);
case MODULE_OP_RMMOD:
    return module_op_rmmod(s,msg,mod_op);
}
return SOS_OK;
```

This code again causes our checker to warn about the memory leak:

```
Warning: Alloced data from instruction #line 186 "loader.c"
mod_op = (sos_module_op_t *) ker_msg_take_data(msg);
is not stored
```

Clearly `mod_op` is still being leaked if `mod_op->op` does not find a matching case in the `switch` statement. Further, adding the `sos_release` attribute to the third formal parameter of the functions `module_op_insmod` and `module_op_rmmod` and rerunning the checker reveals that both of these functions also leak `mod_op`!

A day later, eight weeks after the memory leaks were first introduced, these memory leaks were found and fixed:

```
sos_module_op_t *mod_op;
if (msg->saddr == ker_id() || s->pend) {
    return SOS_OK;
}
mod_op = (sos_module_op_t*) ker_msg_take_data(msg);
if(mod_op == NULL) return -ENOMEM;
switch(mod_op->op){
case MODULE_OP_INSMOD:
    return module_op_insmod(s,msg,mod_op);
case MODULE_OP_RMMOD:
    return module_op_rmmod(s,msg,mod_op);
}
ker_free(mod_op);
return SOS_OK;
```

As shown above, a call to `ker_free` has been added before the final `return`, in order to properly dispose of `mod_op`. The functions `module_op_insmod` and `module_op_rmmod` similarly free their third argument. The CVS log message simply reads "fixed another memory leak."

5 Related Work

As mentioned earlier, the SOS operating system [19] includes ownership annotations that are used to dynamically track memory ownership. These annotations tell the kernel which module owns each block of dynamically allocated memory. When a module is removed from the running system, the kernel automatically deallocates all the memory that the module owns. However, ownership annotations in SOS are *trusted*. For example, missing annotations can cause the kernel's information to become out of date. Also, there are no checks that owners and nonowners meet their obligations. For example, nothing prevents a module from allocating memory and neglecting to eventually free it. Our work grew out of a desire to prevent these kinds of errors from occurring in SOS applications.

Other sensor network platforms have included support for static checking to prevent other classes of errors. For example, the nesC [13] language for sensor networks employs a whole-program analysis to statically detect race conditions. As another example, the galsC [7, 8] language for embedded systems employs an analysis to ensure the type safety of connections between components.

A complementary approach to improving the reliability of sensor network software is through new language abstractions. For example, researchers have explored language support for component-based programming [21, 13, 8], region abstractions [28, 27], component composition [15], and programming in the aggregate [24, 18]. New language constructs enable easier expression of certain programming idioms. They can also make programs more amenable to static checking. Our tool currently analyzes ordinary C programs, since that is the language that SOS employs, but it would be interesting to explore ways to leverage specialized language constructs to improve the tool's effectiveness.

Recent work in the programming languages community has explored the concept of *ownership types* [10, 9, 5, 2] for object-oriented languages. Ownership types designate an owner object for each object, and the static type system ensures a form of *confinement* for each object with respect to its owner. For example, a typical invariant guaranteed by ownership type systems is that an object will only be accessed by its owner or by other objects owned by the same owner. In this way, an object's owner forms a dynamic scope within which the object is confined. Related work on *confined types* [25, 17] provides a more static form of confinement, in which an object is guaranteed not to escape a particular static scope.

Our work provides a static notion of ownership analogous to that of confined types: all accesses to a given resource may only occur within the static scope of its owning module. On a technical level, however, the foundation of our work is quite distinct from that of both ownership types and confined types, as we rely on dataflow analysis rather than on type systems. Our use of dataflow analysis is necessary in order to safely accommodate dynamic transfer of ownership, which the systems described above lack. Ownership transfer is critical in practice for sensor network applications, for example to properly account for split-phase operations. Although recent work has explored a form of transfer in the context of ownership type systems [4], that work requires programmers to provide detailed assertions about ownership, and these assertions are proven as part of a more general program specification and verification framework.

Finally, there have been several proposals for a form of *unique* or *linear* pointer [6, 2, 26, 12], which is guaranteed to be the only reference to its referent. These systems typically include a form of transfer of uniqueness from one pointer to another. For example, a `unique` pointer in AliasJava [2] can be transferred as long as a dataflow analysis shows that the original pointer is no longer accessed after the transfer. Our tool allows a resource to have any number of aliases from within its owner, which is less restrictive than the uniqueness requirement. At the same time, transfer is still allowed safely, as long as none of these aliases are accessed after the transfer.

6 Conclusion

Systems programming APIs for networked embedded systems applications are getting more expressive as the applications become more sophisticated. This necessitates a programming environment that supports program development by providing automated compile-time checkers for proper API usage. Such verification support is particularly important in this domain. First, programmers are often domain experts who may not be systems programming experts. Second, the cost of fixing a bug in the field is high. Third,

networked embedded systems come with somewhat limited debugging support, making it extremely difficult to reason about the cause for a failure in the field.

This paper is our first step toward providing static analysis tools that capture common programming idioms in networked embedded systems. We have focused on dynamic resource management initially, since our experience with SOS suggests that improper memory management is a common source of hard-to-debug problems. We have demonstrated how our checker can effectively find memory errors even in code written by systems experts. Our memory checker is now provided as part of the SOS development environment; it is invoked automatically in the build process.

In the future, we would like to augment the techniques in this paper with support for reasoning about application-level protocols [3, 20] as well as concurrency issues. Ultimately, our goal is to provide a systems programming environment where many common classes of bugs are automatically detected at compile time. We believe such early feedback will help increase the productivity of sensor-network programmers and the reliability of their software.

References

[1] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. Mantis: system support for multimodal networks of in-situ sensors. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 50–59. ACM Press, 2003.

[2] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 311–330. ACM Press, 2002.

[3] Rajeev Alur, Pavol Cerny, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 98–109, New York, NY, USA, 2005. ACM Press.

[4] Anindya Banerjee and David A. Naumann. State based ownership, reentrance, and encapsulation. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 387–411. Springer, 2005.

[5] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230. ACM Press, 2002.

[6] John Boyland. Alias burying: Unique variables without destructive reads. *Software Practice and Experience*, 31(6):533–553, May 2001.

[7] Elaine Cheong, Judy Liebman, Jie Liu, and Feng Zhao. Tinygals: a programming model for event-driven embedded systems. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 698–704, New York, NY, USA, 2003. ACM Press.

[8] Elaine Cheong and Jie Liu. galsc: A language for event-driven embedded systems. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1050–1055, Washington, DC, USA, 2005. IEEE Computer Society.

[9] David G. Clarke, James Noble, and John M. Potter. Simple ownership types for object containment. In *Proceedings of the 2001 European Conference on Object-Oriented Programming*, LNCS 2072, pages 53–76, Budapest, Hungary, June 2001. Springer-Verlag.

[10] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 48–64. ACM Press, 1998.

[11] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier, 1990.

[12] Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 13–24. ACM Press, 2002.

[13] 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 *PLDI 2003: Programming Languages Design and Implementation*, pages 1–11. ACM, 2003.

[14] David Gay and Philip Levis. NesC - References across components, August 2005. Thread on the TinyOS mailing list.

[15] Ben Greenstein, Eddie Kohler, and Deborah Estrin. A sensor network application construction kit (SNACK). In John A. Stankovic, Anish Arora, and Ramesh Govindan, editors, *SenSys*, pages 69–80. ACM, 2004.

[16] Ben Greenstein, Alex Pesterev, Christopher Mar, Eddie Kohler, Jack Judy, Shahin Farshchi, and Deborah Estrin. Collecting high-rate data over low-rate sensor network radios. Technical Report CENS-TR-55, University of California, Los Angeles, Center for Embedded Networked Computing, 2005.

[17] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 241–255. ACM Press, 2001.

[18] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming wireless sensor networks using kairos. In Viktor K. Prasanna, S. Sitharama Iyengar, Paul G. Spirakis, and Matt Welsh, editors, *DCOSS*, volume 3560 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2005.

[19] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, pages 163–176. ACM Press, 2005.

[20] T.A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *FSE 05: Foundations of Software Engineering*, pages 31–40. ACM, 2005.

[21] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS 2000: Architectural Support for Programming Languages and Operating Systems*, pages 93–104. ACM, 2000.

[22] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[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 *CC 02: Compiler Construction*, Lecture Notes in Computer Science 2304, pages 213–228. Springer, 2002.

[24] Ryan Newton and Matt Welsh. Region streams: functional macroprogramming for sensor networks. In *DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks*, pages 78–87, New York, NY, USA, 2004. ACM Press.

[25] Jan Vitek and Boris Bokowski. Confined types. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 82–96. ACM Press, 1999.

[26] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, April 1990.

[27] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *NSDI*, pages 29–42. USENIX, 2004.

[28] Kamin Whitehouse, Cory Sharp, David E. Culler, and Eric A. Brewer. Hood: A neighborhood abstraction for sensor networks. In *MobiSys*. USENIX, 2004.