Ownership is Theft: Experiences Building an Embedded OS in Rust

Amit Levy[†], Michael P Andersen[‡], Bradford Campbell[§], David Culler[‡], Prabal Dutta[§], Branden Ghena[§], Philip Levis[†] and Pat Pannuto[§]

[†]Stanford University {levya,pal}@stanford.edu

[‡]University of California, Berkeley {m.andersen,culler}@berkeley.edu

[§]University of Michigan {bradjc,prabal,brghena,ppannuto}@umich.edu

ABSTRACT

Rust, a new systems programming language, provides compile-time memory safety checks to help eliminate runtime bugs that manifest from improper memory management. This feature is advantageous for operating system development, and especially for embedded OS development, where recovery and debugging are particularly challenging. However, embedded platforms are highly event-based, and Rust's memory safety mechanisms largely presume threads. In our experience developing an operating system for embedded systems in Rust, we have found that Rust's *ownership* model prevents otherwise safe resource sharing common in the embedded domain, conflicts with the reality of hardware resources, and hinders using closures for programming asynchronously. We describe these experiences and how they relate to memory safety as well as illustrate our workarounds that preserve the safety guarantees to the largest extent possible. In addition, we draw from our experience to propose a new language extension to Rust that would enable it to provide better memory safety tools for event-driven platforms.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Design, Security

Keywords

Rust, Linear Types, Ownership, Embedded Operating Systems

1. INTRODUCTION

Safe languages promise to eliminate a large class of programming errors at compile time. Safety is particularly appealing for an operating system kernel. Safety makes buffer and integer overflows, which constitute a significant fraction of kernel bugs [7], impossible. Strongly typed languages can isolate different components of a system from each other at a finer granularity than hardware protection mechanisms, which impose a high context switching overhead when used at a fine grain (e.g., per device driver [12]).

The benefits of language safety are especially attractive for an embedded operating system kernel, for three reasons. First, embedded systems often use processors and microcontrollers that lack hardware protection mechanisms such as memory management units: language safety can protect software when hardware cannot. Second, embedded applications are less tolerant of crashes, as they cannot rely on user intervention to recover from runtime errors (e.g., restart the application). Third, debugging embedded kernels is especially difficult because they often do not have logging features and require physical access to attach a debugger. While safety does not prevent logical errors or bugs, it does greatly protect an embedded kernel from hard crashes.

Unfortunately, the downsides of additional features typical with safe languages often outweigh these benefits for systems programming. Garbage collection, for example, introduces nondeterministic delays. Automatic memory allocators complicate common kernel optimizations such as slab allocation [5].

Rust [2], a new, safe language designed for systems programming, including operating system kernels, promises memory safety with no runtime overhead. Rust differs from most safe languages in that it maintains safety and speed without having a garbage collector. It relies heavily on compile-time checks that detect data races and unsafe memory accesses at no run-time overhead.

We have been using Rust to develop a new embedded operating system for microcontrollers called Tock. In particular, Tock targets embedded platforms with much less than a megabyte of memory—for example, our development platform has 64KB of RAM. At first examination, Rust seems perfectly suited for this task. Rust achieves memory and type safety without garbage collection by using a mechanism, derived from affine types and unique pointers, called ownership. However, in our experience so far, ownership semantics have introduced new and unexpected challenges developing our operating system.

In our development efforts, we have encountered three problems with using Rust to implement an embedded kernel. First, Rust's automatic memory management is not optimized for hardware resources and device drivers that are always present in the system. Second, Rust's ownership model prevents resource sharing between closures and other kernel code, due to unnecessary thread safety concerns in our setting. Finally, although closures are desirable for simplified event handling, their requirement for dynamic memory is problematic for embedded systems.

We describe each of these problems, illustrating examples where Rust issues occur. We have been able to mostly workaround these issues. However, in many cases this came at the cost of incorporating more of the operating system into the trusted computing base, which is allowed to circumvent the type system.

To better enable Rust to support event-driven embedded platforms, we explore a possible language feature we call *execution contexts*. This feature would provide Rust with a valuable tool for allowing safe memory sharing when underlying hardware constraints or execution models can reliably prevent concurrency issues.

2. RUST

Rust offers two important features that make it attractive as a

PLOS'15, October 04-07 2015, Monterey, CA, USA

ACM 978-1-4503-3942-1/15/10 ...$15.00

DOI: http://dx.doi.org/10.1145/2818302.2818306.

language for writing a secure, embedded operating system. First, it preserves type safety without relying on a runtime garbage collector for memory management. Instead, Rust uses affine types [14] to determine when memory can be freed *at compile-time*. Second, similar to Modula-3 [6] and Haskell [13], Rust allows the programmer to explicitly separate code which is strictly bound to the type syste from code which may subvert it.

2.1 Memory Management with Ownership

Most safe languages achieve memory safety by using runtime checks to determine when memory can safely be freed. Rust, instead, avoids the runtime overhead by using the concept of *ownership*, generally referred to as affine types in the literature [14].

Each value in Rust has a unique *owner*—namely, the variable to which it is bound. When the owner of a value goes out of scope, the value is freed. For example:

```
{
  let x = 43;
}
```

Within the scope above, memory for the value `43` is allocated and bound to the variable `x`. When the scope exits and `x` is no longer accessible its memory is reclaimed.

Because there can only be a single owner, aliasing is disallowed. Instead, values are either copied (if the type of the value implements the `Copy` trait) or moved between variables. Once a value is moved, it is no longer accessible from the original variable binding. For example, the following code is not permissible:

```
{
  let x = Foo::new();
  let y = x;
  println("{}", x);
}
```

Because `Foo::new()` has been moved from `x` to `y`, `x` is no longer valid. Similarly, moving a value into a function by passing it as an argument invalidates the original variable. As a result, functions must explicitly hand ownership back to the caller:

```
fn bar(x: Foo) -> Foo {
  // do something useful with ‘x‘
  x   // <- return x
}

let my_x = Foo;
let my_x = bar(my_x);
```

To simplify programming, Rust allows references to a value, called *borrows*, without invalidating the original variable. Borrows are created using an `&` and can be either mutable or immutable. There are two main restrictions on borrows:

1. A value can only be mutably borrowed if there are no other borrows of the value.

2. Borrows cannot outlive the value they borrow. This prevents dangling pointer bugs.

This ownership model allows the compiler to provide two important safety mechanisms. First, it allows the compiler to determine when to free dynamically allocated memory from the stack or heap. Memory bugs like double-free and use-after-free are impossible in this model. Second, it eliminates many data races by preventing concurrent access to resources by multiple threads.

In many systems, this ownership model works well. For example, in a threaded network server, resources such as client requests are logically isolated from each other and can be owned by a single thread. When multiple threads need to share a resource, they can pass ownership through channels. However, as we discuss in the rest of this paper, it does not work well in systems that use non thread-based concurrency models and when resources must be shared.

2.2 The `unsafe` Keyword

Rust has an explicit separation between *trusted* code, which can circumvent the type system in certain ways, from *untrusted* code, which is strictly bound to the type system. Specifically, trusted code can use blocks wrapped in the `unsafe` to perform unsafe operations (e.g. dereference a raw pointer) or call other unsafe functions.

The `unsafe` keyword can be used in two ways. First, any block of code can be wrapped in an unsafe block to allow it to perform operations that might break the type system. For example, a hardware abstraction layer can use this feature to expose a memory-mapped I/O register as a normal Rust struct:

```
let mydevice : &mut IORegs = unsafe {
  &mut *(0x200103F as *mut IORegs)
}
```

Second, functions can be annotated with `unsafe` which prevents untrusted code from calling them. For example, the standard library's `transmute` function casts its input into any other type of the same size:

```
pub unsafe fn transmute<T,U>(e: T) -> U
```

Packages compiled with `-F unsafe_code` cannot use the `unsafe` keyword, allowing systems builders to isolate trusted code on a per package basis. An operating system fundamentally must perform certain operations which violate the type system. For example, hardware is often configured through memory mapped I/O registers which must be cast into usable data structures from arbitrary pointers. This mechanism allows the operating system to separate between trusted modules which must, for example, address hardware directly, and untrusted modules like device drivers which should access hardware through a narrower interfaces. When developing a secure system, the challange lies in minimizing the amount of code that uses unsafe language features.

3. CHALLENGES

Resource constrained microcontrollers face challenges in terms of energy use, memory availability, and execution time that uniquely shape the operating system design.

In particular, embedded operating systems must be more reliable and use less memory than their general-purpose counterparts. For example, the platform which Tock targets has only 64KB of memory. Most embedded platforms do not typically exceed 256KB of RAM, and most have significantly less. As a result, embedded operating systems avoid memory intensive mechanisms like threading, and instead opt for event driven concurrency [10].

Similarly, microcontrollers generally do not have hardware memory management units and cannot support virtual memory. As a result, they avoid dynamic memory mechanisms both because swapping memory to gracefully degrade upon memory exhaustion is not possible, and because dynamic allocations may be explicitly prohibited (e.g. for MISRA C [11] compliance). Conversely, average case performance is not generally a bottleneck, so embedded operating systems can trade CPU cycles to save memory or provide safety.

Reliability is paramount for embedded operating systems due to the difficulty of debugging and lack of human interaction. Leveraging the compile time guarantees of a safe language to isolate kernel extensions can help make the operating system more reliable without restricting flexibility. The SPIN [4] operating system,

for example, was written in Modula-3. SPIN showed how a safe language provides safe, fast access to protected kernel resources. This safe access, in turn, enables applications to safely extend the kernel with new features and services. Having similar safe extensibility is extremely valuable in an embedded kernel, because a kernel must be able to adapt to many custom hardware configurations and applications.

Implementing an operating system for an embedded platform in Rust, which is nominally well suited for these limitations as it is a low-level language that provides memory safety, has raised three main issues: the tension between ownership and cyclic dependencies in an operating system, capturing state in callback closures, and statically allocating closure memory.

3.1 Resource Ownership

Rust's memory ownership model, while providing useful memory-management tools, can conflict with common scenarios that arise in operating systems. First, many operating system resources are not dynamically allocated. For example, hardware peripherals are always present, and device drivers are allocated once at system initialization. As these resources are never freed, the ownership mechanisms do not need to manage the lifetimes of these resources.

Second, resources must often be shared between multiple logical units. For example, consider a simple networked application where a reference to the networking stack must be shared between a UDP interface (to allow applications to send packets) and the underlying radio driver (to allow the radio hardware to notify that packets have been received).

```
// Both UDP and RadioDriver need a reference to the
// networking stack.

impl UDP {
  // Called from an application to send a packet. Must be able
  // to tell the networking stack to transmit a new packet.
  fn send(&mut self, packet) {
    self.network_stack.outgoing(packet);
  }
}

impl RadioDriver {
  // Called from the main loop when a packet arrives.
  // on_receive needs a reference to network_stack
  // to notify it of a received packet.
  fn on_receive(&mut self, packet) {
    self.network_stack.incoming(packet);
  }
}
```

In order to avoid possible data races, Rust's ownership model does not allow the UDP interface and RadioDriver to keep references to the networking stack simultaneously. While hardware interrupts are asynchronous, and therefore run concurrently with other kernel code, in our operating system interrupt handlers enqueue tasks to be run in the main scheduler loop, which is single-threaded. As a result, `on_receive` and `send` can never run concurrently and no data race is possible.

We overcome both issues by declaring most resources as static variables and borrowing them with `'static` lifetime. In Rust, borrows with `'static` lifetimes are treated specially: only unsafe code can create such a borrow, but once created they can be re-borrowed mutably any number of times. Intuitively, the `'static` lifetime points to the "original sin" of borrowing a static variable unsafely. Therefore, none of the safety semantics that apply to normal borrows are applied to those with static lifetimes.

Being able to mutably borrow static variables multiple times means our operating system can allow multiple drivers to reference hardware resources or each other. However, doing so means we lose the ability to detect real data races at compile time, for example, if the kernel passes a shared resource to an interrupt handler that may run concurrently. Instead, we resort to an overly conservative convention in which interrupt handlers perform no work except posting to the main scheduler's task queue. An ideal mechanism would, instead, allow for shared state between components that will never run concurrently, but disallow sharing when they may.

3.2 Callbacks through Closures

Embedded operating systems are largely event-driven: most computation happens in response to events—e.g. a timer firing, an I/O operation completing, or an external interrupt triggering. The standard C approach to event-driven code is to pass a callback function pointer as a parameter to an asynchronous call. When the asynchronous operation completes, it invokes the callback. There are two problems with this approach. First, a linear sequence of asynchronous operations does not appear in a linear piece of code. Instead, it is spread across a series of many small functions. Second, the programmer must manually pass state between callers and callbacks, e.g., by using "stack ripping" [3].

Event-driven application languages, such as JavaScript, help solve this problem by allowing callbacks to be specified as closures at the call site:

```
var count = 0;
setInterval(function() {
  console.log(count + " clicks");
}, 2000);
onClick(function() {
  count += 1;
});
```

The ability write to the callback at the point it is registered makes reading asynchronous code easier. Furthermore, the ability to close over variables from the caller makes resource management simpler.

However, this programming approach requires closure to capture shared state to avoid stack ripping. In the example above, when the call to `onClick` returns, there are three valid handles to `count`: one in the caller, and one in each of the closures. In the case of JavaScript, this is safe: programs are single-threaded, so the callback cannot access the closure while the calling context executes.

Rust has closures, but does not assume a particular threading model, so its ownership system prohibits a similar construction. For the closure to capture a variable, it must either take ownership of it, preventing the caller from accessing it, or complete before returning to the caller.

```
let mut x = 0;
setInterval(move || {  // "||" indicates a closure in Rust
  println!("{} clicks", x);
}, 2000);
// x is no longer valid in this context, and we
// cannot create the closure for onClick
```

In the context of an embedded OS, this can be critically limiting. In practice, this leads software to take one of two approaches, neither desirable. The first carefully partitions state between a caller and callbacks. For example, instead of using a single, intuitive variable for the LEDs on a device that allows all of the lights to be controlled:

```
// No other code can access LEDs
setTimeout(|| {
  leds.activityToggle();
}, 2000);
```

the ownership concerns force the code to partition the LEDs so that access to each one is stored in separate variables.

```
setTimeout(|| {
  activityLed.toggle();
}, 2000);
```

Unfortunately, the end result of partitioning is tiny, fine-grained interfaces that are very hard to manage. A callback that needs to toggle multiple LEDs, for example, needs to capture each one separately in the closure.

The second approach is to avoid compile time ownership checks and rely on run-time mechanisms. While this may work for some applications, it defeats the purpose of leveraging compile-time safety checks for an embedded operating system.

3.3 Closure Memory Management

Since closures capture dynamic variables, and several instances of a closure may be outstanding concurrently, closures are typically allocated dynamically. However, embedded operating systems often do not, or cannot support dynamic memory allocation.

On the other hand, several common asynchronous patterns do not require closures to be re-entrant, and the memory for the closure can instead be statically allocated at the call site. For example, in TinyOS [10], a callback is either posted or not—posting it multiple times results in a single invocation. In Rust, it makes sense to express this with closures:

```
fn set_lcd<F: Fn()>(text: [u8; 256], on_done: &F) {
    spi_write(START, to_static(|| {
        spi_write(text, onDone);
      })
    );
}
```

The `set_lcd` function writes a `START` sequence to the SPI bus connected to an LCD screen, followed by the text to display on the screen. Multiple outstanding closures will never exist as another SPI command cannot be executed until the current command is completed. The goal of `to_static` in the example above is to copy captured values into a statically allocated memory buffer (in this case the `text` buffer and a pointer to the `on_done` callback). But how would we implement `to_static`?

In Rust, each closure has a unique type. Therefore, the following `to_static` signature would result in a separate instantiation of the function for each closure passed to it:

```
fn to_static<F: Fn()>(closure: F) -> &F
```

In principal, this would allow us to statically allocate a buffer sized for each particular closure. Unfortunately, there is no `size_of` keyword in Rust, and static initialization cannot invoke functions—in this case the function `size_of<T>() -> usize` provided by the Rust core library. In comparions, it is possible to write a version of `to_static` in C++11, which has lambdas with similar semantics to Rust closures, *as well as* a `size_of` keyword that is resolved at compile-time.

While the syntactic additions required to support statically allocating closures in Rust would be small, they would have a profound difference for engineering resource constrained systems. In our embedded operating system, we have had to abandon closures as a callback mechanism, in favor of more cumbersome, and less comprehensible styles such as statically binding callbacks and manual stack ripping.

4. PROPOSED LANGUAGE MECHANISM: EXECUTION CONTEXTS

The core issue underlying hardware resource sharing and callback closures is that Rust does not allow mutable aliasing. This helps prevent common bugs such as data races between threads, iterator invalidation etc. However, under certain constraints, for example, if all aliases are used fromthe same thread, mutable aliases might be perfectly safe. This is the common case for embedded systems, which typically have only one primary execution thread occasionally punctuated by interrupt handlers. Currently, however, Rust's type system is not rich enough to express constraints on thread execution.

We propose an extension to the Rust type-system, called *execution contexts*, that reflects the thread of a value's owner in its type. We argue that this type information could be used to allow multiple borrows of a value from within the same thread, but not across threads. Execution contexts are a compile-time only mechanism that allow the compiler to identify safe operations.

To define an execution context, type parameters are prefixed with the hash character (`#`), akin to current lifetime annotations (`'`). A value's execution context is determined by its owner. When a value is moved between threads (e.g. through a channel), it takes on the execution context of its new owner. Borrows have a *borrowed execution context* (in addition to their own execution context as values) that reflects the execution context of the value they borrow. Finally, execution contexts can be instantiated as type parameters in functions and trait implementations. For example, closure traits (e.g., `FnMut`, `FnOnce`) have an execution context parameter that determines the context in which they run. Table 1 summarizes the different types that may include an execution context. While every reference has an execution context, like lifetimes they can be elided in most cases.

Value	`#a val`
Borrow	`& #a var`
Closure Trait	`FnOnce<#a>`
Function Parameter	`fn f<#a>(var: & #a T) -> #a ()`

Table 1: Execution context usage. An execution context `a` is represented in the type system as `#a`. Execution contexts can be attached to values, borrows, and closure traits, and can also be used as parameters in functions.

4.1 Semantics of Execution Contexts

The goal of execution contexts is to allow programs to mutably borrow values multiple times as long as those borrows are never shared between threads. Execution contexts allow the compiler to distinguish such sharing from actual errors using only local analysis.

By definition, the whole execution of a scope must run in one execution context. Thus every value owned in that scope must have the same execution context. Because callers must own the return value of their callees, values in a parent and a child scope must have the same execution context. Therefore, a *thread* (a single call-graph with no internal concurrency) maps to an execution context.

Entry functions (e.g. *main* or *extern* functions such as signal or hardware interrupt handlers) begin a new execution context. Functions can also "create" new execution contexts by forcing closures to run in a different execution context than the return value of the function. For example, the `spawn` function from the standard library, which runs its argument in a new POSIX thread:

```
fn spawn<#a, #b, F>(func: F) -> #a ()
  where F: FnOnce() + #b;

// Does not compile: Cannot borrow across contexts
let x = 0;
spawn(|| { println!("In thread: {}", x); });

// Does compile: Can take sole ownership with move
```

```
let x = 0;
spawn(move || { println!("In thread: {}", x); });
```

The return value of `spawn` has execution context `#a`, meaning the caller of `spawn` has that execution context. Conversely, the argument to `spawn` is a closure that runs with execution context `#b`. Because `#a` might not equal `#b`, the closure cannot borrow values from the caller, protecting against possible race conditions.

Borrowed execution contexts reflect the original value's execution context and *do not* change when the borrow is moved. Furthermore, because a value cannot be moved while there are borrows of it, a borrowed execution context always matches the execution context of the original value. While borrows, themselves, can be moved between threads, they can only be dereferenced from an execution context that matches their borrowed execution context. Importantly, this allows borrows to be stored as fields in data structures that might be moved between threads, while preserving thread safety.

Execution contexts permit code that was always safe but previously invalid in Rust. Consider a ballot box that stores votes as a list of closures to run when votes are tallied:[1]

```
struct BallotBox<'a, #a> {
  votes: LinkedList<HeapPtr<FnMut<#a>() + 'a>>,
  yays: #a u32
}

impl<'a, 'b: 'a, #a> BallotBox<'a, #a> {
  pub fn vote_yay(&'b #a mut self) -> #a () {
    votes.push_back(
        HeapPtr::alloc(|| {self.yays += 1})
    )
  }
  pub fn tally(& #a mut self) -> #a u32 {
    for f in self.votes.iter_mut() {
      f();
    }
    return self.yays;
  }
}
```

The stored closures each hold a mutable reference to `self` with the same execution context (`#a`) as constrained by the type parameter initialization from the `impl` statement. Since the closures are run to completion in the same thread, these multiple borrows do not introduce race conditions.

In some cases, it is necessary to permit multiple execution contexts to access the same values. Figure 1 shows how an interrupt handler, which executes in a unique handler context (`#h`), can post a task to a shared queue that the scheduler, executing the main kernel context (`#k`), can dequeue and run. This example introduces the last new construct: the `#any` execution context. Any execution context may access a value with the `#any`. Accesses must be serialized internally, for example, using mutexes or atomic sections (disabling interrupts) as appropriate.

4.2 Limitations

Multi-threaded execution is not the only issue that arises from mutable aliasing. For example, internal references union types may break the type system if different types may point to overlapping memory [1]. Similarly, dynamically sized data structures such as vectors must not free data that may still be referenced by a different alias. Therefore, supporting mutable aliasing in Rust might require subtle changes to the standard library. While we believe execution contexts can, in general, be safe, we have not fully explored their implications on the wider Rust ecosystem.

[1]We use `HeapPtr` types to allocate heap memory. Readers familiar with Rust should read these as Rust's `Box` type.

```
// InterruptQueue is shared between threads. It is marked with
// the #any context to signify that it can be mutably borrowed
// from any thread. The enqueue and dequeue methods must
// be implemented using atomic blocks for safety
let interrupt_queue: #any InterruptQueue =
    InterruptQueue::new();

struct ButtonDriver<#k> {
  count: #k usize
}

impl<#k> ButtonDriver<#k> {
  pub fn handle_interrupt(& #k mut self) {
    // Because self is borrowed with thread #k, we know
    // the caller is in the same thread that owns self.count,
    // and are able to borrow it mutably. In our case, this is
    // because 'handle_interrupt' was called from main
    self.count += 1;
  }
  pub fn RAW_INTERRUPT<#h>(& #h mut self) {
    // Cannot borrow self.counter because it belongs to
    // context #k and not #h. Instead self is enqueued to
    // be handled from the main thread
    interrupt_queue.enqueue(self);
  }
}

fn main() {
  let button_driver = ButtonDriver{count: 0};
  loop {
    interrupt_queue.dequeue().handle_interrupt();
  }
}
```

Figure 1: Complete example of execution contexts. An `interrupt_queue` is shared between a lightweight "top-half" interrupt handler that runs in the interrupt context and the (naïve) scheduler that executes the majority of the interrupt logic in the "bottom-half" interrupt handler in the main kernel execution context.

5. RELATED WORK AND CONCLUSIONS

Several previous operating systems have used language features to guarantee the safety of kernel components. SPIN [4] allows applications to download extensions written in Modula-3 into the kernel, and uses the language to sandbox the extensions. Singularity [9] requires that applications as well as the entire kernel are written in a managed language (C#) and relies entirely on a language sandbox (rather than hardware protection) to isolate applications. Unlike Singularity, Tock cannot rely on a garbage collected language due to the constraints of embedded systems. TinyOS [10] detects data races between threads at compile time and requires the programmer to wrap accesses to shared data in atomic blocks that temporarily disable interrupts. However, TinyOS uses a superset of nesC (a C dialect) which provides no memory or type safety.

Clarke and Wrigstad [8] point out a similar issue arising from borrowing unique types in a closed loop. They propose the notion of *external uniqueness*, which allows an object subgraph to contain cyclic internal references while having a unique external owner. Our proposal for *execution contexts* in Rust is similar but considers execution threads, instead of object graphs, as the unit of isolation.

Rust's low-level interface, safe memory management, and large community make it a particularly good fit for operating system development. If future language development can address the challenges we have demonstrated, Rust should be well positioned to support the next generation of correct embedded operating systems.

6. REFERENCES

[1] Rust issue: "borrowck is unsound in the presence of &'static

muts". https://github.com/rust-lang/rust/issues/27616.

[2] The Rust programming language. http://www.rust-lang.org.

[3] ADYA, A., HOWELL, J., THEIMER, M., BOLOSKY, W. J., AND DOUCEUR, J. R. Cooperative task management without manual stack management. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2002), ATEC '02, USENIX Association, pp. 289–302.

[4] BERSHAD, B. N., CHAMBERS, C., EGGERS, S., MAEDA, C., MCNAMEE, D., PARDYAK, P., SAVAGE, S., AND SIRER, E. G. Spin–an extensible microkernel for application-specific operating system services. *ACM SIGOPS Operating Systems Review 29*, 1 (1995), 74–77.

[5] BONWICK, J., ET AL. The slab allocator: An object-caching kernel memory allocator. In *USENIX summer* (1994), vol. 16, Boston, MA, USA.

[6] CARDELLI, L., DONAHUE, J., JORDAN, M., KALSOW, B., AND NELSON, G. The modula– type system. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1989), POPL '89, ACM, pp. 202–212.

[7] CHEN, H., MAO, Y., WANG, X., ZHOU, D., ZELDOVICH, N., AND KAASHOEK, M. F. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (2011), ACM, p. 5.

[8] CLARKE, D., AND WRIGSTAD, T. External uniqueness is unique enough. *ECOOP 2003–Object-Oriented Programming* (2003), 59–67.

[9] HUNT, G. C., AND LARUS, J. R. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review 41*, 2 (April 2007), 37–49.

[10] LEVIS, P., MADDEN, S., POLASTRE, J., SZEWCZYK, R., WHITEHOUSE, K., WOO, A., GAY, D., HILL, J., WELSH, M., BREWER, E., AND CULLER, D. TinyOS: An operating system for sensor networks. In *Ambient Intelligence*, W. Weber, J. Rabaey, and E. Aarts, Eds. Springer Berlin Heidelberg, 2005, pp. 115–148.

[11] MOTOR INDUSTRY SOFTWARE RELIABILITY ASSOCIATION, ET AL. *MISRA-C: 2012: Guidelines for the Use of the C Language in Critical Systems*. MIRA, 2013.

[12] SWIFT, M. M., MARTIN, S., LEVY, H. M., AND EGGERS, S. J. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop* (2002), ACM, pp. 102–107.

[13] TEREI, D., MARLOW, S., PEYTON JONES, S., AND MAZIÈRES, D. Safe haskell. In *Proceedings of the 2012 Haskell Symposium* (New York, NY, USA, 2012), Haskell '12, ACM, pp. 137–148.

[14] TOV, J. A., AND PUCELLA, R. Practical affine types. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2011), POPL '11, ACM, pp. 447–458.