The Case for Writing a Kernel in Rust

Amit Levy
Stanford University
levya@cs.stanford.edu

Bradford Campbell
University of Michigan
bradjc@umich.edu

Branden Ghena
University of Michigan
brghena@umich.edu

Pat Pannuto
University of Michigan
ppannuto@umich.edu

Prabal Dutta
University of Michigan
prabal@umich.edu

Philip Levis
Stanford University
pal@cs.stanford.edu

ABSTRACT

Decades of research has attempted to add safety mechanisms to operating system kernels, but this effort has failed in most practical systems. In particular, solutions that sacrifice performance have been generally avoided. However, isolation techniques in modern languages can provide safety while avoiding performance issues. Moreover, utilizing a type-safe language with no garbage collector or other runtime services avoids what would otherwise be some of the largest sections of trusted code base. We report on our experiences in writing a resource efficient embedded kernel in Rust, finding that only a small set of unsafe abstractions are necessary in order to form common kernel building blocks. Further, we argue that Rust's choice to avoid runtime memory management by using a linear type system will enable the next generation of safe operating systems.

ACM Reference format:
Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. The Case for Writing a Kernel in Rust. In *Proceedings of APSys '17, Mumbai, India, September 2, 2017,* 7 pages.
https://doi.org/10.1145/3124680.3124717

1 INTRODUCTION

Most operating system kernels assume all kernel code is trusted. In part, this is because systems builders have traditionally relied on hardware enforced memory protection, and the process abstraction in particular, to provide isolation. The process, however, is a heavy-weight abstraction: it has stacks, file descriptors, and many kernel data structures. Furthermore, switching between virtual address spaces is costly. Systems such as Nooks [23], lightweight contexts [15] and seL4 [12] have all explored using address spaces for isolation at a finer grain than a process, showing ways to significantly reduce overhead.

This paper takes a more extreme approach: entirely throw away hardware protection within the kernel and, instead, write the kernel in a memory-safe programming language. This approach has been tried before, with varying success: Spin [3] allows applications to extend and optimize kernel performance by downloading modules written in Modula-3 [6], while Singularity [9] is written in Sing# (a variant of C#) and provides a software isolated process (SIP) abstraction. Both Spin and Singularity, however, use garbage-collected languages, which pose many problems for kernels. Garbage collection complicates memory placement and layout, creates timing non-determinism from background locks and introduces stop-the-world intervals that pause the entire OS. Furthermore, both Spin and Singularity depend on a large, unsafe code base: the Spin kernel and Singularity's runtime, respectively.

Encouraged by recent advances in type-safe programming languages, this paper proposes, instead, to write the entire kernel in a type-safe programming language that is *not* garbage collected. By not relying on a complex garbage collector or other runtime services, such a kernel is simultaneously memory safe *and* gives kernel programmers the degree of memory management control and deterministic behavior they need. A recent language, Rust [17], fits this model. In Rust, values are tracked by their lifetime and deallocated as they go out of scope.

While writing any kernel requires some unsafe code, we argue that a primary design goal for kernels is to *minimize the amount of unsafe code that must be trusted.* We built a kernel for low-power uniprocessors that follows this design goal. In our kernel, unsafe code falls into two categories: Rust library code, written by language developers, and kernel code, written by kernel developers. The required Rust library code includes only a few low-level operations, such as floating point, bounds checks, and type casts. The kernel

APSys '17, September 2, 2017, Mumbai, India

ACM ISBN 978-1-4503-5197-3/17/09…$15.00
https://doi.org/10.1145/3124680.3124717

trusted code is also extremely small. It includes the standard abstractions of context switches, system call traps, interrupt handling, and memory-mapped I/O, as well as one new abstraction, `TakeCell`, that stems from Rust's memory model. A `TakeCell` allows kernel code to safely modify complex structures by using inline closures that statically compile with no overhead.

We describe Rust's memory model and the challenges kernel code introduces to this model [14] (Section 2). We then describe a minimal set of trusted abstractions that we use to build a kernel (Section 3) and provide a few examples of how our kernel uses these abstractions to provide common OS features (Section 4).

2 RUST

Rust is a type-safe language designed for systems software [17]. Originally, it was motivated by the challenges of writing Firefox's layout engine to be both fast and highly parallel. Since then, it has also been successfully used in large scale projects like Dropbox's back end storage [18]. It is a particularly attractive language for low-level systems because it preserves type-safety (e.g. no memory leaks or buffer overflows) while providing runtime characteristics similar to C's. This section gives a brief overview of the challenges that Rust's memory management poses to writing a kernel; interested readers can refer to Levy et al. [14] for more details.

2.1 Only One Mutable Reference

Rust uses a concept called *ownership* (an affine type system [24]) to determine at *compile time* when memory should be freed. The principal challenge that ownership introduces to kernel software is that there cannot be two mutable references (non-const pointers in C) to the same memory [14]. This is necessary because allowing mutable aliases would allow a program to circumvent the type system [8]. For example, consider Rust's `enum` types which allow multiple distinct types to share the same memory, similar to unions in C. In this example, the `enum` can be either a 32-bit unsigned number, or a mutable reference (pointer) to a 32-bit unsigned number:

```
// Rust
enum NumOrPointer {
  Num(u32),
  Pointer(&mut u32)
}
```

```
// Equivalent C
union NumOrPointer {
  uint32_t Num;
  uint32_t* Pointer;
};
```

Unlike unions in C, a Rust `enum` is type safe. The language ensures that it is impossible to access a `NumOrPointer` as a `Num` when the compiler thinks it is a `Pointer`, and vice-versa.

Having two mutable references to the same memory could violate `NumOrPointer`'s safety and would allow code to construct arbitrary pointers and access any memory. Suppose

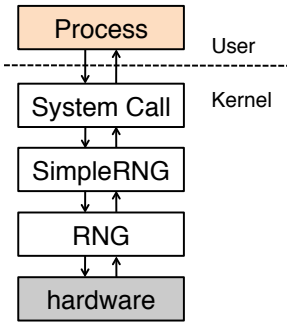

Figure 1: Software architecture for a system call interface to a hardware random number generator: both `RNG` and the system call interface need references to `SimpleRng`.

that the `NumOrPointer` is currently a `Pointer`. If one of the references is to a `Pointer` but the other can change it to a `Num`, then it can create an arbitrary pointer:

```
// Rust
// n.b. will not compile
let external : &mut NumOrPointer;
match external {
  Pointer(internal) => {
    // This would violate safety and
    // write to memory at 0xdeadbeef
    *external = Num(0xdeadbeef);
    *internal = 12345;  // Kaboom
  },
  ...
}

// Equivalent C
// compiles without warning
union NumOrPointer* external;
uint32_t* numptr = &external->Num;
*numptr = 0xdeadbeef;
*external->Pointer = 12345;
```

2.2 Kernels Need Multiple References

Operating system kernels depend heavily on callbacks and other event-driven programming mechanisms. Often, multiple components must both be able to mutate a shared data structure. Consider, for example, the random number generator software stack as shown in Figure 1. `RNG` provides an abstraction of an underlying hardware RNG, such as Intel's RDRAND/RDSEED [10] or a TRNG on an ARM processor [1].

`SimpleRng` sits between `RNG` and the system call layer. It translates between userspace system calls and the `RNG` interface. It calls into `RNG` when a process requests random numbers and calls back to the system call layer when random numbers are ready to deliver to the process. A natural way of structuring this stack is for both the system call layer and `RNG` to have a reference to `SimpleRng`:

```
pub struct SimpleRNG {
  busy: bool,
  ...
}

impl SimpleRng {
  fn command(&mut self) { self.busy = true; ... }
  fn deliver(&mut self, rand: u32) { self.busy = false; ... }
}

impl SysCallDispatcher {
  fn dispatch(&mut self, num: u32) {
    match num {
      // ...
      43 => self.simple_rng.command(),
    }
  }
}

impl RNG {
  fn done(&mut self, rand: u32) {
    self.simple_rng.deliver(rand);
  }
}
```

Rust's ownership model does not allow both structures to have a mutable reference to `SimpleRng` (the reference must be mutable because `command` marks `SimpleRNG` as `busy`, and `deliver_rand` marks it as not `busy`, mutating the internal state of the `SimpleRNG` object). Prior work suggested that this problem means writing a kernel requires changes to the Rust language [14]. However, the next section describes an alternate solution, consisting of a minimal set of trusted code that includes two software abstractions, `Cell` and `TakeCell`.

3 TOWARDS A RUST KERNEL

Writing an operating system in Rust does not require any changes to the language, but does require trusting two types of unsafe code. The first consists of Rust language mechanisms and libraries, written by the Rust language team. These provide a safe interface, but their underlying implementations use unsafe code. The second type of code which must be trusted are portions of kernel code, written by kernel developers, that use unsafe code in order to implement basic operating system functionality but again can provide safe interfaces.

The rest of this section describes what code falls into these two categories. Together, they constitute the complete set of unsafe code in the kernel and are surprisingly small, primarily consisting of mechanisms that any language or kernel needs to provide. There are two additional abstractions, however. One provided, by Rust, is a `Cell`, which allows multiple references to a mutable object but exposes a limited API that requires copying the data out in order to access it, rather than accessing it in place. The second, provided by the kernel, is called *TakeCell*. A *TakeCell* allows kernel code to safely share complex structures with no runtime overhead (such as copies) and a very simple programming abstraction.

3.1 Trusted Rust Code

Rust has a large set of available libraries, including data structures, web browser engines, JavaScript compilers and I/O. A kernel requires only *libcore*, which supports primitive types such as arrays and an interface to compiler (LLVM [13]) intrinsic operations. A kernel wanting to use a generic, existing dynamic memory allocator would require *liballoc* as well.

Both of these libraries contain some trusted code either because they must subvert the type system (memory management requires type casts) or for performance optimizations. Specifically, a kernel relies on the following four Rust abstractions that use unsafe code:

Bounds checks: Arrays are bounds-checked, so unsafe code uses the length field to ensure accesses are safe.

Iterator optimizations: The canonical way to operate across Rust arrays is with iterators, which use unsafe code to avoid unnecessary intermediate checks.

Compiler intrinsics and primitive casts: Floating point, volatile loads/stores and casting between primitive types have architecture specific details that Rust relies on LLVM for.

Cell: An abstraction that encapsulates data such that interior references cannot escape and it can be operated on with an immutable reference.

`Cell` provides a partial solution to the problem of event driven code needing to hold multiple mutable references (Section 2.2). A Rust `Cell` is an opaque memory container that code can copy into and out of, but cannot internally reference. The key feature of `Cell` is that an immutable reference can copy into it. The unsafe type cast that arose with enums in Section 2.1 cannot happen with `Cell` since multiple referrers operate on separate copies of the shared data.

```
pub struct SimpleRng {
  busy: Cell<bool>,
  ...
}

impl Syscall for SimpleRng {
  fn command(&self) {
    ...
    self.busy.set(true);
  }
}
```

Above, we show how `Cell` can solve the problem in the random number generator example (Figure 1). Both the `RNG` and the system call dispatcher hold an immutable reference to the same `SimpleRng`. Normally, this would mean that

calls from either would not be able to modify SimpleRNG's internal state. However, as shown below, Cell allows the command method, to set busy to be true even though &self is an *immutable* reference.

Cell, unfortunately, is only a partial solution. It imposes the significant cost of requiring memory copies, which is an unacceptable overhead for complex or large kernel data structures. The next section describes a new abstraction, TakeCell, which allows safe, efficient implementations of complex kernel abstractions.

3.2 Trusted Kernel Code

The Rust abstractions described above all provide safe interfaces to the kernel programmer. Assuming these abstractions are correctly implemented, they do not allow callers to violate type safety. Kernels, however, must do some fundamentally unsafe things (such as context switch), and must wrap these unsafe implementations in safe interfaces. Surprisingly, this requires very little Rust code. The following six pieces of unsafe kernel code need to be trusted:

Context switches: Switching between thread contexts requires saving and restoring the program counter and stack pointer as well as potentially changing process state between protected and unprotected mode.

Memory-mapped I/O and structures: Processors provide I/O through memory-mapped registers, so the kernel needs to be able transform raw memory addresses (e.g., 0x40008000) into typed registers and bit fields, while file systems require casting disk blocks into structures.

Memory allocator: Kernels define specialized memory allocators (e.g., slab [4]) which must type cast raw pointers.

Userspace buffers: Because user space could pass invalidly sized buffers to the kernel, unsafe code must check that buffers are valid.

Interrupt/exception handlers: Handlers are inherently unsafe because they preempt running code so memory may be in an inconsistent state.

TakeCell: An abstraction that allows multiple references, like Cell, but without memory copies.

TakeCell is unique among these in that it is a purely software abstraction designed to allow efficient, safe kernel code. Cell works well for primitive types and small values for which the copying semantics do not add any overhead. On these types, Cell optimizes down to just loading into a register and storing to memory. TakeCell is for larger or more complex data objects. Rather than copy values out of a TakeCell, a program passes code *in*, through a closure. For example, a system call interface which keeps caller/process state through a structure named App can access it this way:

```
struct App { /* many variables */ }
app: TakeCell<App>

self.app.map(|app| {
  // code can read/write app's variables
});
```

Like Rust's Cell, a TakeCell internally owns the shared data, e.g. a mutable reference, and a TakeCell can be shared by multiple callers. However, rather than copy values out of a TakeCell, its API consists of a single method, map(f). Normally, map(f) will invoke the closure f with a reference to the TakeCell's internal data. However, in cases where there already exists a reference to the internal data—such as a recursive call—map(f) is a no-op and does not execute the closure.

As a result, TakeCell is a form of mutual exclusion. Like a mutex, TakeCell ensures that there is only one mutable reference to the internal value. However, unlike a mutex, it skips the operation instead of blocking. Once compiled, TakeCell is just as fast as unchecked C code. For example, the following snippet of TakeCell code

```
struct App {
    count: u32,
    tx_callback: Callback,
    rx_callback: Callback,
    app_read: Option<AppSlice<Shared, u8>>,
    app_write: Option<AppSlice<Shared, u8>>,
}
pub struct Driver {
    app: TakeCell<App>,
}

driver.app.map(|app| {
    app.count = app.count + 1
});
```

generates the following ARM assembly, which safely checks if app is a null pointer, operates on app.count only if it is a value, then stores the result:

```
/* Load App address into r1, replace with null */
ldr     r1, [r0, 0]
movs    r2, 0
str     r2, [r0, 0]
/* If TakeCell is empty (null) return */
cmp     r1, 0
it      eq
bx      lr
/* Non-null: increment count */
ldr     r2, [r1, 0]
add     r2, r2, 1
str     r2, [r1, 0]
/* Store App back to TakeCell */
str     r1, [r0, 0]
bx      lr
```

Note that, since the closure is scoped to the stack frame, it requires no special allocation.

4 CASE STUDIES

In this section, we briefly describe three kernel abstractions in Rust, showing how given the special cases described in Section 3, kernel building blocks are naturally expressed in Rust. These cases are taken from our kernel. The kernel's trusted computing base includes the Rust core library as well as under 1000 lines out of over 6000 lines of kernel code.

4.1 Direct Memory Access

Direct memory access (DMA) is a common source of such violations in kernels today. Because the hardware will use whatever address it is given, kernel code using DMA can circumvent virtual memory and other protection mechanisms [11].

A Rust-based kernel exposes memory-mapped registers as typed data structures. Exposing them safely in this manner ensures that kernel code cannot write arbitrary values to them. For example, a DMA interface that uses a Rust slice (dynamically sized array)

```
struct DMAChannel {
  ...
  enabled: Cell<bool>,
  buffer: TakeCell<&'static mut [u8]>,
}
```

enforces that the `buffer` field is a valid pointer to a block of memory. Furthermore, it can use the buffer length to ensure it does not write past the end of the block. For a caller to pass a `&'static [u8]`, it must have been granted access to a statically allocated byte buffer. This interface also highlights an interesting safety concern that Rust enforces. Rust cannot reason about how long the DMA operation will take, but it needs assurances that the buffer will still be live (not freed) when it completes. The only types of memory that can satisfy this are statically (global) allocated buffers and heap buffers, so it requires that the buffer is `'static`. The only unsafe code in the DMA implementation is the code that memory-maps registers to write the buffer and length into the DMA registers and enables the completion interrupt.

4.2 Universal Serial Bus

Universal Serial Bus (USB) uses in-memory *descriptors* specified by the programmer to configure and control USB endpoints. The hardware assumes these descriptors are laid out in a particular way. Representing these hardware memory structures in Rust is straight-forward.

This example below shows a hardware interface that relies on two levels of data structures that require a particular layout and reference integrity. `USBRegisters.in_endpoints` is reference to an array of endpoint descriptors. `InEndpoint` lays out exactly how the processor lays out these descriptors in memory. Finally, the `EpCtl` type defines the set of valid values which are checked at compile-time. It is worth noting that unlike pointers, references cannot be null; an `InEndpoint` can exist if and only if `dma_address` points to a valid `DMADescriptor` and a `USBRegisters` can exist if and only if `in_endpoints` points to a valid array of 16 `InEndpoints`.

```
enum EpCtl {
  ...
  Enable = 1 << 31,
  ClearNak = 1 << 26,
  Stall = 1 << 21
}
struct InEndpoint {
  control: Cell<EpCtl>,
  dma_address: Cell<&'static DMADescriptor>,
  ...
}
struct USBRegisters {
  ...
  // There can be 16 endpoints
  in_endpoints: Cell<&[InEndpoint; 16]>,
  ...
}
```

4.3 Complex Data Structures

It is common for kernel components, such as the buffer cache, page tables, and file systems to rely on data structures with circular references like doubly linked-lists or trees. This often requires multiple aliases to the same mutable data, but those aliases can be logical. For example, a buffer cache entry references a disk block, but this is encoded as a device id and sector number, and so does not require managing Rust's ownership semantics. Cases where data structures use bi-directional pointers such as doubly-linked lists can be handled with the same principles used for circular dependencies between kernel components: `Cell` and `TakeCell`. For example:

```
struct ListLink<T>(Cell<Option<&T>>);
struct BufferHead {
  state: BufferState,
  next: &ListLink<BufferHead>,
  prev: &ListLink<BufferHead>,
  page: &Page,
  ...
}
```

4.4 Multicore

For sake of simplicity and brevity, this paper has examined using Rust in a single-threaded setting. Supporting multicore systems requires managing concurrency within the kernel. Rust was originally designed for parallel systems. As a result, it has language mechanisms that allow the programmer to safely manage concurrency. For example, `Sync` specifies that a structure can safely be shared across threads and `Channels`, a mechanism to pass data across threads, only admits `Sync`

types. A multicore Rust kernel would use such mechanisms to maintain safety of shared data across cores.

5 FUTURE WORK

Our work so far discusses kernel mechanisms and drivers pertinent to a kernel for low-power uniprocessor applications such as USB applications and DMA. We believe other operating systems components, such as file systems or video buffers would use similar techniques, though actual implementation efforts will likely uncover more challenges. In particular, while we show that in-memory data structures like linked lists and trees can be modeled safely in Rust, future work should explore how to do so for data structures on disk or in hardware (e.g. the page table).

Moreover, we did not evaluate our design in a multi-processor setting. Systems in other type-safe languages, like Singularity, have successfully modeled concurrent computing in the language. Future work should explore how to avoid growing the trusted computing base in service of concurrency.

Finally, the ability to write a low-overhead kernel in a type-safe language gives way to the potential for novel applications in security and concurrency. While there has been extensive research in both areas, they have seldom been applied to kernels. In particular, we believe a promising area of future work is Dynamic Information Flow Control.

Information flow control (IFC) [20] is a security mechanism that enforces non-interference by labeling inputs and outputs with security labels, and tracks the propagation of labels throughout the system. Typically, systems built on IFC either use static labels [16, 19], which must be specified at compile-time, or minimize resource consumption by sacrificing granularity [7, 25]. Recently, promising work [5, 21, 22] has leveraged purity in functional languages like Haskell to get the best of both worlds: dynamic IFC labels with minimal memory overhead.

Enforcing dynamic information flow control in a Rust kernel is an exciting prospect. However, while Balasubramanian et al [2] briefly proposed a static IFC language based on Rust, it is not yet clear if such implementations would be sound, and even less clear if linear types are sufficient to enforce dynamic IFC in lieu of purity or an effects system. Future work should explore whether these primitives can be enforced at the language level using Rust and, if so, how it impacts kernel design.

6 CONCLUSION

Decades of research have attempted to add safety mechanisms to operating system kernels, but this effort has failed in most practical systems. Language-only techniques can mitigate the performance and granularity issues arising from hardware enforced memory isolation. Moreover, using a type-safe language with no garbage collector or other runtime services, such as Rust, avoids what would otherwise be one of the largest sections of trusted code base. While previous efforts to use such languages for kernel development have concluded that changes to the language would be required, we find that the language is sufficient and only a small set of unsafe abstractions are necessary to form common kernel building blocks, hopefully enabling the next generation of safe operating systems.

REFERENCES

[1] Atmel. *ARM SAM4L Low Power MCU*, 3 2014. 4203G.

[2] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamaric, and L. Ryzhyk. System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS'17, Berkeley, CA, USA, 2017. USENIX Association.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 267–283. ACM, 1995.

[4] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 6–6, Berkeley, CA, USA, 1994. USENIX Association.

[5] P. Buiras, D. Vytiniotis, and A. Russo. Hlio: Mixing static and dynamic typing for information-flow control in haskell. *SIGPLAN Not.*, 50(9):289–301, Aug. 2015.

[6] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The Modula–3 type system. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 202–212, New York, NY, USA, 1989. ACM.

[7] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association.

[8] D. Grossman. Existential types for imperative languages. In *Proceedings of the 11th European Symposium on Programming Languages and Systems*, ESOP '02, pages 21–35, London, UK, UK, 2002. Springer-Verlag.

[9] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, Apr. 2007.

[10] Intel. *IntelÂő Digital Random Number Generator Software Implementation Guide*, 8 2012. Rev. 1.1.

[11] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 314–327, Washington, DC, USA, 2006. IEEE Computer Society.

[12] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[13] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[14] A. Levy, M. P. Andersen, B. Campbell, D. Culler, P. Dutta, B. Ghena, P. Levis, and P. Pannuto. Ownership is theft: Experiences building an embedded OS in Rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, PLOS '15, pages 21–26, New York, NY, USA, 2015. ACM.

[15] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel. Light-Weight Contexts: An OS abstraction for safety and performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 49–64, GA, 2016. USENIX Association.

[16] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association.

[17] N. D. Matsakis and F. S. Klock, II. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA, 2014. ACM.

[18] C. Metz. The Epic Story of Dropbox's Exodus From the Amazon Cloud Empire. https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/.

[19] A. C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.

[20] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, Oct. 2000.

[21] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in haskell. *SIGPLAN Not.*, 44(2):13–24, Sept. 2008.

[22] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in haskell. *SIGPLAN Not.*, 46(12):95–106, Sept. 2011.

[23] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 102–107, New York, NY, USA, 2002. ACM.

[24] J. A. Tov and R. Pucella. Practical affine types. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 447–458, New York, NY, USA, 2011. ACM.

[25] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. *Commun. ACM*, 54(11):93–101, Nov. 2011.