System Programming in Rust: Beyond Safety

Abhiram Balasubramanian*
University of Utah

Marek S. Baranowski
University of Utah

Anton Burtsev
UC Irvine

Aurojit Panda
UC Berkeley

Zvonimir Rakamarić
University of Utah

Leonid Ryzhyk
VMware Research

ABSTRACT

Rust is a new system programming language that offers a practical and safe alternative to C. Rust is unique in that it enforces safety without runtime overhead, most importantly, without the overhead of garbage collection. While zero-cost safety is remarkable on its own, we argue that the superpowers of Rust go beyond safety. In particular, Rust's linear type system enables capabilities that cannot be implemented efficiently in traditional languages, both safe and unsafe, and that dramatically improve security and reliability of system software. We show three examples of such capabilities: zero-copy software fault isolation, efficient static information flow analysis, and automatic checkpointing. While these capabilities have been in the spotlight of systems research for a long time, their practical use is hindered by high cost and complexity. We argue that with the adoption of Rust these mechanisms will become commoditized.

ACM Reference format:
Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond Safety. In *Proceedings of HotOS '17, Whistler, BC, Canada, May 08-10, 2017,* 6 pages.
https://doi.org/10.1145/3102980.3103006

1 INTRODUCTION

For several decades system developers choose C as the one and only instrument for programming low-level systems. Despite many advances in programming languages, clean-slate operating systems [3], hypervisors [2], key-value stores [26], web servers [30], network [6] and storage [38] frameworks are still developed in C, a programming language that is in many ways closer to assembly than to a modern high-level language.

Today, the price of running unsafe code is high. For example, in 2017, the Common Vulnerabilities and Exposures database lists 217 vulnerabilities that enable privilege escalation, denial-of-service, and other exploits in the Linux kernel [8], two-thirds of which can be attributed to the use of an unsafe language [5]. These include human mistakes related to low-level reasoning about intricate details of object lifetimes, synchronization, bounds checking, etc., in a complex, concurrent environment of the OS kernel. Even worse, pervasive use of pointer aliasing, pointer arithmetic, and unsafe type casts keeps modern systems beyond the reach of software verification tools.

Why are we still using C? The historical reason is performance. Traditionally, safe languages rely on managed runtime, and specifically garbage collection (GC), to implement safety. Despite many advances in GC, its overhead remains prohibitive for systems that are designed to saturate modern network links and storage devices. For example, to saturate a 10Gbps network link, kernel device drivers and network stack have a budget of 835 ns per 1K packet (or 1670 cycles on a 2GHz machine).With the memory access latency of 96-146 ns [28], the I/O path allows a handful of cache misses in the critical path—the overhead of GC is prohibitive.

Is it reasonable to sacrifice safety for performance, or should we prioritize safety and accept its overhead? Recent developments in programming languages suggest that this might be a false dilemma, as it is possible to achieve both performance and safety without compromising on either. The breakthrough has been achieved through synthesis of an old idea of *linear types* [41] and pragmatic language design, leading to the development of the Rust language [18]. Rust enforces type and memory safety through a restricted ownership model, where there exists a unique reference to each live object in memory. This allows statically tracking the lifetime of the object and deallocating it without a garbage collector. The runtime overhead of the language is limited to array bounds checking, which is avoided in most cases by using iterators.

In this paper, we strengthen the case for Rust as a systems programming language by demonstrating that its advantages go *beyond safety*. We argue that Rust's linear type system enables capabilities missing in traditional programming languages (both safe and unsafe). We identify three categories of such capabilities: *isolation*, *analysis*, and *automation*.

Isolation. Software fault isolation (SFI) enforces process-like boundaries around program modules in software, without relying on hardware protection [42]. While SFI improves the security and reliability of the system, it is hard to implement efficiently. Existing SFI implementations do not support sending data across protection boundaries by reference, as this enables the sender to maintain access to the data. Hence a copy is required to ensure isolation, making such solutions unacceptable in line-rate systems. Rust's single ownership model allows us to implement zero-copy SFI. The Rust compiler ensures that, once a pointer has been passed across isolation boundaries, it can no longer be accessed by the

*Work performed at Samsung Reserch America.

HotOS '17, May 08-10, 2017, Whistler, BC, Canada

ACM ISBN 978-1-4503-5068-6/17/05…$15.00
https://doi.org/10.1145/3102980.3103006

sender. Our SFI implementation (section 3) introduces the overhead of 90 cycles per protected method call and has zero runtime overhead during normal execution.

Analysis. Programming languages literature describes a number of language extensions and associates static analyses that enforce security and correctness properties beyond traditional type safety [9, 29, 39]. For example, *static information flow control (IFC)* enforces confidentiality by tracking the propagation of sensitive data through the program [29]. Many of these analyses are complicated by the presence of aliasing, e.g., in the IFC case, writing sensitive data to an object makes this data accessible via all aliases to the object and can therefore change the security state of multiple variables. Modern alias analysis achieves efficiency by sacrificing precision, posing a major barrier to accurate IFC. By restricting aliasing, Rust sidesteps the problem. We illustrate this in section 4 by prototyping an IFC extension for Rust based on precise, yet scalable program analysis.

Automation. Many security and reliability techniques, including transactions, replication, and checkpointing, internally manipulate program state by traversing pointer-linked data structures in memory. Doing so automatically and for arbitrary user-defined data types can be complicated in the presence of aliasing. For example, during checkpointing, the existence of multiple references to an object may lead to the creation of multiple object copies (Figure 3). Existing solutions require for a developer to write checkpointing code manually or modify the application to use special libraries of checkpointable data structures. In section 5, we propose a Rust library that adds the checkpointing capability to arbitrary data structures in an efficient and thread-safe way.

The features discussed above have been in the spotlight of systems research for decades; however their practical adoption is hindered by their high cost and complexity. *We argue that with the adoption of Rust as a systems programming language, these mechanisms will become commoditized.* We support our thesis by discussing a prototype implementations of SFI, IFC, and checkpointing in Rust.

The advantages of Rust come at the cost of learning a new language and porting software to it, dealing with a limited and evolving Rust ecosystem, and increased design complexity due to having to comply with Rust's restricted ownership model. We believe that these overheads are justified in applications that require uncompromised safety and performance. In fact, we argue that forcing the developer to be explicit about resource ownership is a good practice in system programming. At the same time, Rust is clearly not an optimal language for rapid prototyping, scripting, and other non-performance-critical tasks. We hope that observations we make in this paper will help Rust find its proper place in the system programmer's toolkit.

2 BACKGROUND

The design of Rust builds on a body of research on linear types [41], affine types, alias types [43], and region-based memory management [40], and is influenced by languages like Sing# [16], Vault [17] and Cyclone [23]. Prior to Rust, the Singularity OS [16] introduced linear types to systems research. Singularity's Sing# language supports a hybrid type system, where traditional types are used to enforce software fault isolation between processes by allocating each process its own garbage-collected heap. Linear types are used exclusively for zero-copy inter-process communication via a shared exchange heap. In section 3, we present a solution that achieves both isolation and communication using linear types.

Rust supports automatic memory management without a garbage collector through its *ownership* model. When a variable is *bound* to an object, it acquires ownership of the object. The object is deallocated when the variable goes out of scope. Alternatively, ownership can be transferred to another variable, destroying the original binding. It is also possible to temporarily *borrow* the object without breaking the binding. Visibility of the borrow is restricted to the syntactic scope where it is declared and cannot exceed the scope of the primary binding. The following code snippet illustrates Rust's ownership model:

```
fn take(v: Vec<i32>) //captures ownership of v.
fn borrow(v: &Vec<i32>)//`&' denotes borrowed object
let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];
take(v1);
//Error: binding v1 was consumed by take()
println!("{:?}", v1);
borrow(&v2);
// OK: binding v2 is preserved by borrow()
println!("{:?}", v2);
```

Single ownership eliminates pointer aliasing, making it impossible to implement data structures like doubly-linked lists in pure Rust. The language offers two mechanisms to remedy this limitation. First, Rust embeds an *unsafe* subset that is not subject to the single ownership restriction and is used to, e.g., implement parts of Rust's standard library, including linked lists. Techniques described in this paper rely on properties of the Rust type system that only hold for the safe subset of the language. In the rest of the paper we assume that unsafe code is confined to trusted libraries. Second, Rust supports safe read-only aliasing by wrapping the object with a reference counted type, `Rc` or `Arc`. When write aliasing is essential, e.g., to access a shared resource, single ownership can be enforced *dynamically* by additionally wrapping the object with the `Mutex` type. In contrast to conventional languages, this form of aliasing is explicit in the object's type signature, which enables us to handle such objects in a special way as described in section 5.

With the maturing of Rust, we now can apply linear types to a broad range of systems tasks. Multiple projects have demonstrated that Rust is suitable for building low-level high-performance systems, including an embedded [25] and a conventional OS [10], a network function framework [31], and a browser engine [35]. While these systems primarily take advantage of Rust's type and memory safety, effectively using it as a safe version of C, we focus on the capabilities of Rust that go beyond type and memory safety.

Similar in spirit to ours is the work by Jespersen et al. [22] on session-typed channels for Rust, which exploits linear types to enable compile-time guarantees of adherence to a specific communication protocol. There is a large body of research on safe system programming languages, including safe dialects of C [9, 23] as well as alternatives such as Go and Swift. While a comparison of Rust against these languages is outside the scope of this paper, we point out that, unlike Rust, all of them rely on runtime support

for automatic memory management, either in the form of garbage collection or pervasive reference counting.

3 ISOLATION

We argue that Rust enables software fault isolation (SFI) with lower overhead than any mainstream language. SFI encapsulates untrusted extensions in software, without relying on hardware address spaces. While modern SFI implementations enable low-cost isolation of, e.g., browser plugins [44] and device drivers [15], their overhead becomes unacceptable in applications that require high-throughput communication across protection boundaries. Consider, for instance, network processing frameworks such as Click [24] or NetBricks [31], which forward packets through a pipeline of filters. Security and fault tolerance considerations call for isolating each pipeline stage in its own protection domain. The traditional SFI architecture achieves this by confining memory accesses issued by the isolated component to its *private heap* [15, 19, 44]. Sending data across protection boundaries requires copying it, which is unacceptable in a line-rate system. An alternative architecture [27] uses a shared heap and tags every object on the heap with the ID of the domain that currently owns the object. This avoids copying, but introduces a runtime overhead of over 100% due to tag validation performed on each pointer dereference.

Rust enables SFI without copying or tagging. Type safety provides a foundation for SFI by ensuring that a software component can only access objects obtained from the memory allocator or explicitly granted to it by other components. In addition, Rust's single ownership model enforces that, after passing an object reference to a function or channel, the caller loses access to the object and hence can neither observe nor modify data owned by other components (with the exception of safe read-only sharing allowed by Rust).

What is missing for a complete SFI solution is a management plane to control domain lifecycle and communication by cleaning up and recovering failed domains, enforcing access control policies on cross-domain calls, etc. We demonstrate how such mechanisms can be implemented in Rust as a library. Our implementation is straightforward, as it relies on inherent capabilities of Rust. The significance of our result is that it provides a constructive proof that Rust enables fault isolation, including secure communication across isolation boundaries, with negligible overhead. To the best of our knowledge, *this is the first SFI implementation in any programming language to demonstrate these properties.*

Our SFI library exports two data types: *protection domains (PDs)* and *remote references (rrefs).* All PDs use a common heap for memory allocation; however they do not share any data. PDs interact exclusively via method invocations on rrefs. Arguments and return values of remote invocations follow the usual Rust semantics: borrowed references are accessible to the target PD for the duration of the call; all other arguments change their ownership permanently. The sole exception is remote references: the object pointed to by an rref stays in its original domain and can only be accessed from the domain holding the reference via remote invocation.

Rrefs are implemented as smart pointers (Figure 1). When an rref is created, the original object reference is stored in the *reference table* associated with the domain. This reference acts as a proxy for remote invocations. The rref returned to the user contains a

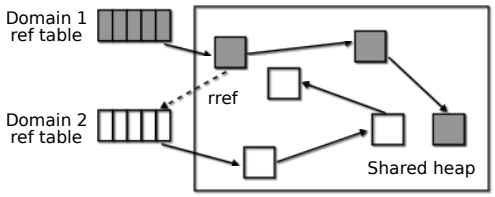


Figure 1: All PDs share the common heap. Cross-domain references (rrefs) are mediated by the reference table.

weak pointer [12] to the reference table. A weak pointer does not prevent the object it points to from being destroyed and must be upgraded to a strong pointer before use. Proxying remote invocations through the reference table gives the owner of the domain complete control over its interfaces and lifecycle. For example, they can intercept remote invocations for fine-grained access control or revoke a remote reference completely by removing its proxy from the reference table. In the latter case, future attempts to invoke the rref will fail to upgrade the weak pointer and will return an error. The following listing illustrates the use of domains and rrefs:

```
/* Inside domain manager: */
let d = Domain::new(); // create a PD
//create an object inside PD and wrap it in RRef
let rref = Domain::execute(&d,
            ||RRef::new(createSomeObj()));
...
/* Invoke rref from another PD: */
match rref.method1() {
  Ok(ret) => println!("Result: {}", ret),
  Err(_)  => println!("method1() failed") }
```

By clearing the reference table one can automatically deallocate all memory and resources owned by the domain. We use this mechanism to implement fault recovery. When a panic occurs inside the domain (e.g., due to a bounds check or assertion violation), we first unwind the stack of the calling thread to the domain entry point [11] and return an error code to the caller. Next, we clear the domain reference table and finally run the user-provided recovery function to re-initialize the domain from clean state. The recovery process can re-populate the reference table, thus making the failure transparent to clients of the domain.

Our SFI implementation introduces the overhead of indirect invocation via the proxy. In addition we use thread-local store [7] to store ID of the current protection domain. We evaluate this overhead in the context of the NetBricks network function framework [31] running on an 8-core Intel Xeon E5530 2.40GHz server. NetBricks is implemented in Rust and performs on par with optimized C frameworks. It retrieves packets from DPDK [6] in batches of user-defined size and feeds them to the pipeline, which processes the batch to completion before starting the next batch. Batches are passed between pipeline stages via function calls. NetBricks takes advantage of linear types to ensure that only one pipeline stage can access the batch at any time. While Panda et al. [31] refer to this mechanism as fault isolation, NetBricks does not support fault containment or recovery.

We use our SFI library to isolate every pipeline component in a separate protection domain, replacing function calls with remote invocations. We measure the cost of isolation by constructing a pipeline of null-filters, which forward batches of packets without doing any work on them. We vary the length of the pipeline and the number of packets per batch, and measure the average number

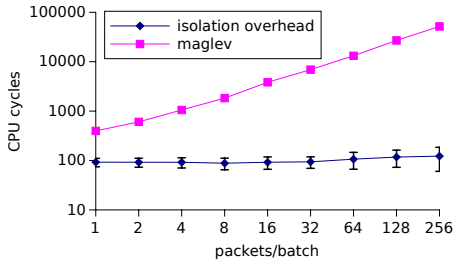


Figure 2: Overhead of remote invocation for different batch sizes plotted against the cost of processing by Maglev.

of cycles to process a batch with and without protection. The difference between the two divided by the pipeline length gives us the overhead of a remote invocation over regular function call. We found this overhead to be independent of the pipeline length, and hence Figure 2 shows the results for the length of 5. The overhead grows from 90 CPU cycles for 1-packet batches to 122 cycles for 256-packet batches, which is roughly the cost of 2 or 3 L3 cache accesses. We attribute the increase to higher cache pressure due to retrieving more packets from DPDK. To put these numbers in perspective, we compare them against the cost of batch processing in a realistic, but light-weight, network function—the NetBricks implementation of the Maglev load balancer [13], which was shown to perform competitively with an optimized C version [31]. The overhead of isolation is negligible (under 1%) for batches larger than 32 packets (Figure 2). Finally, we measure the cost of recovery by simulating a panic in the null-filter and measuring the time it takes to catch it, clean up the old domain, and create a new one. The recovery took 4389 cycles on average.

4 ANALYSIS

We argue that Rust enables precise and efficient static information flow control (IFC). IFC enables strong security guarantees for untrusted modules by ensuring that they do not leak sensitive data through unauthorized channels [29, 45]. To this end, program inputs are assigned security labels. Program output channels are also assigned labels, which bound the confidentiality of data sent through the channel. The compiler or the verifier tracks the flow of sensitive data through the program by tainting the result of each expression with the upper bound of labels of its arguments , ultimately proving that the program respects channel bounds. This check must be performed statically to avoid the overhead of runtime validation and to prevent leaks arising from the program paths not taken at run time.

We illustrate the ideas behind IFC with an example implementation of a buffer that provides methods to append to and print its content:

```
1 struct Buffer{data: Option<Vec<u8>>}
2 impl Buffer {
3   fn new() -> Buffer {Buffer{data: None}}
4   fn append(&mut self, mut v: Vec<u8>) {
5     match self.data {
6       None => self.data = Some(v),
7       Some(ref mut d) => d.append(&mut v) }
8   } }
```

The following program creates an empty buffer and appends a secret and non-secret value to it (we introduce a new kind of annotation to Rust to attach security labels to variables). It then attempts to print the content of the buffer:

```
9  let mut buf = Buffer::new();
10 #[label(non-secret)] //security annotation for IFC
11 let nonsec = vec![1,2,3];
12 #[label(secret)] //security annotation for IFC
13 let sec = vec![4,5,6];
14 buf.append(nonsec);
15 buf.append(sec); // buf now contains secret data
16 println!("{:?}",buf.data);//ERROR:leaks secret data
17 //println!("{:?}",nonsec);
```

The `println!()` macro outputs data to an untrusted terminal; therefore it only allows non-secret arguments (the corresponding annotation is not shown). The program violates this constraint by writing sensitive data to the store and then attempting to print it out. This violation can be detected via efficient static analysis, which tracks the flow of data to and from the buffer. Specifically, in line 15, the content of the buffer is tainted as secret, which triggers an error in line 16.

In conventional programing languages, information flow analysis is complicated by pointer aliasing. We demonstrate this by attempting to introduce a more subtle vulnerability to the above program in line 17. Note that our buffer implementation uses the first vector of values received from the client to store the data internally (line 6), and later appends new data to it (line 7). We exploit this behavior by writing a non-secret vector to the empty buffer first (line 14), appending secret data to the buffer (line 15), and finally printing out the modified content of the non-secret vector, which now aliases the secret data in the buffer (line 17). Thus, instead of writing sensitive data to an unauthorized output channel directly, the program creates an alias to an object, waits until the object obtains sensitive data *through a different alias*, and then leaks the data via the original alias.

Rust prevents such exploits by design, as they violate single ownership. In this example, line 17 is rejected by the compiler, as it attempts to access the `nonsec` variable, whose ownership was transferred to the `append` method in line 14. In contrast, detecting such leaks in a conventional language requires tracking all pointer aliases and reflecting any change in the security label made via one alias to all others. Alias analysis is undecidable in theory and is hard to perform efficiently in practice without losing precision.

An alternative to alias analysis is a security type system, where an object's type includes its security label that cannot change, making aliasing safe [29]. In our example, we would assign a low-security type to the non-secret vector, and a high-security type to the `buf` variable, prompting the compiler to reject an attempt to write to the latter an alias to the former in line 6. Instead, we must allocate a new vector and copy over the content of the `v` argument. While the type-based approach enables fast compile-time analysis, it introduces the overhead of extra memory allocation and copying, which may not be acceptable in a line-rate system.

By eliminating aliasing, Rust enables efficient static information flow analysis while allowing for security labels to change at runtime. We formulate IFC as the problem of verification of an abstract interpretation of the program [34]. We represent the value of each variable in the abstract domain by its security label. Input variables are initialized with user-provided labels. Arithmetic expressions over secure values are abstracted by computing the upper bound

of their arguments. An auxiliary program counter variable is introduced to track the flow of information via branching on labeled variables. We verify the resulting abstract program to ensure that labels written to output channels do not exceed user-provided channel bounds. Our methodology is similar to that by Zanioli et al. [45], sans the expensive alias analysis step.

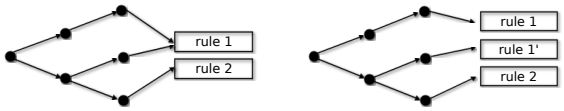
We have implemented a minimal proof-of-concept IFC for Rust. Our prototype relies on Rust macros to transform the program into its abstract interpretation. There currently does not exist a dedicated verifier for Rust. Hence, we extended the SMACK verifier [32] with an early version of the Rust frontend. SMACK verification toolchain is built on top of the LLVM compiler infrastructure that is used by Rust as well. Hence, it was relatively straightforward to extend SMACK with a preliminary support for verification of Rust programs.

We implemented and verified a simple secure data store in Rust, which stores data on behalf of multiple clients, while preventing non-privileged clients from reading data belonging to privileged ones. The security-label bounds were specified in the example program through the use of assertions. As a sanity check, we seeded a bug into checking of security access in the implementation. SMACK discovered the injected bug, thereby increasing our confidence in the verification process. Even without alias analysis, verification can be expensive for large programs. Further improvements can be achieved through compositional reasoning: in the absence of aliasing, the effect of every function on security labels is confined to its input arguments and can be summarized by analyzing the code of the function in isolation from the rest of the program.

5 AUTOMATION

Many techniques for improving the performance and reliability of systems hinge on the ability to automatically manipulate program state in memory. In particular, checkpointing [14], transactions [20, 21, 33], replication [36, 37], multiversion concurrency [1, 4], etc., involve snapshotting parts of program state. This, in turn, requires traversing pointer-linked data structures in memory. Ideally one would like to generate this functionality automatically and for arbitrary user-defined data types. However, doing so in a robust way can be complicated in the presence of aliasing.

Consider, for instance, the task of checkpointing the state of a network firewall that consists of rules indexed via a trie for fast rule lookup based on packet headers (Figure 3a). Multiple leaves of the trie can point to the same rule, causing this rule to be encountered multiple times during pointer traversal, potentially leading to redundant copies of the rule, as shown in Figure 3b. To avoid this in a conventional language, one must record the address of each object reached during the traversal and check newly encountered objects against the recorded set. This has the obvious downside of increasing the CPU and memory overhead of checkpointing. Another complication is related to external pointers aliasing parts of the object. Such pointers, which do not own the data they point to, must be handled in a special way during pointer traversal, for example, a transaction system may add the target of such a pointer to the transaction set. However conventional languages do not provide means to identify such pointers.



(a) Database before checkpointing. **(b) Multiple copies due to naïve traversal of the graph.**

Figure 3: Checkpointing a firewall rule database.

Rust simplifies the problem dramatically: by default, all references in Rust are unique owners of the object they point to and can be safely traversed without extra checks. Aliasing, when present, is explicit in object's type signature: only objects wrapped in reference counted types (`Rc`, `Arc`) can be aliased. The `Rc` and `Arc` wrappers therefore provide a convenient place to deal with aliasing with minimal modifications to user code and without expensive lookups.

To support this observation, we implemented an automatic checkpointing library for Rust. We develop a *trait* (traits are analogous to Java interfaces), called `Checkpointable`, with two methods: `checkpoint()` and `restore()`. We introduce a compiler plugin that inductively generates an implementation of this trait for types comprised of scalar values and references to other checkpointable types. Next, we provide a custom implementation of `Checkpointable` for `Rc` (`Arc` can be extended similarly), which sets an internal flag the first time `checkpoint()` is called on the object and checks this flag to avoid creating additional copies when graph traversal hits the object again via a different alias. Our library adds the checkpointing capability to arbitrary user-defined data types; in particular it checkpoints objects with internal aliases correctly and efficiently.

6 CONCLUSION AND FUTURE WORK

Rust represents a unique point in the language design space, bringing the benefits of type and memory safety to systems that cannot afford the cost of garbage collection. We explore the benefits of Rust that go beyond safety. We show that Rust enables system programmers to implement powerful security and reliability mechanisms like SFI, IFC, and automatic checkpointing more efficiently than any conventional language. This is just the tip of the iceberg: we believe that further exploration of linear types in the context of real systems will yield more game-changing discoveries.

One promising direction is formal verification, both automatic and user-guided. Alias analysis is a major source of complexity and imprecision in software analysis. By lifting the burden of resolving memory aliasing from the verifier, Rust enables faster and more accurate verification. This has numerous applications in systems, ranging from verified kernel extensions to, potentially, fully verified hypervisors, embedded OSs, etc.

ACKNOWLEDGMENTS

We thank the anonymous HotOS reviewers. This material is partially based upon work supported by the National Science Foundation under Grants No. 1319076 and No. 1527526.

REFERENCES

[1] Daniel Atkins, Alex Potanin, and Lindsay Groves. 2013. The Design and Implementation of Clocked Variables in X10. In *Proceedings of the Thirty-Sixth Australasian Computer Science Conference - Volume 135 (ACSC '13)*. Adelaide, Australia, 87–95.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. 2003. Xen and the art of virtualization. In *ACM SIGOPS Operating Systems Review*, Vol. 37. ACM, 164–177.

[3] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. Big Sky, Montana, USA, 29–44.

[4] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. 2010. Concurrent programming with revisions and isolation types. In *ACM Sigplan Notices*, Vol. 45. ACM, 691–707.

[5] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys '11)*. Shanghai, China, Article 5, 5 pages.

[6] Intel Corporation. DPDK: Data Plane Development Kit. http://dpdk.org/. (????).

[7] Alex Crichton. 2017. scoped-tls. https://github.com/alexcrichton/scoped-tls. (2017).

[8] CVE. Vulnerabilities on Linux Kernel Machines. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33. (????).

[9] Robert DeLine and Manuel Fähndrich. 2001. Enforcing High-level Protocols in Low-level Software. In *ACM Conference on Programming Language Design and Implementation (PLDI '01)*. Snowbird, Utah, USA, 59–69.

[10] Redox Project Developers. Redox - Your Next(Gen) OS. (????). http://www.redox-os.org/.

[11] The Rust Project Developers. 2017. Implementation of Rust stack unwinding. https://doc.rust-lang.org/1.3.0/std/rt/unwind/. (2017).

[12] The Rust Project Developers. 2017. Struct std::rc::Weak. https://doc.rust-lang.org/std/rc/struct.Weak.html. (2017).

[13] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. Santa Clara, CA, 523–535.

[14] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)* 34, 3 (2002), 375–408.

[15] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. 2006. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. Seattle, Washington, 75–88.

[16] Manuel Fähndrich et al. 2006. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *Eurosys*.

[17] Manuel Fahndrich and Robert DeLine. 2002. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. Berlin, Germany, 13–24.

[18] Mozilla Foundation. The Rust programming language. https://doc.rust-lang.org/book/. (????).

[19] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. 2002. The JX Operating System. In *USENIX Annual Technical Conference*. Monterey, CA, USA, 45–58.

[20] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*. Chicago, IL, USA, 48–60.

[21] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. 2003. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing (PODC '03)*. Boston, Massachusetts, 92–101.

[22] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session Types for Rust. In *Workshop on Generic Programming*.

[23] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference (ATEC '02)*. Monterey, CA, USA, 275–288.

[24] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *ACM Trans. Comput. Syst.* 18, 3 (Aug. 2000), 263–297.

[25] Amit Levy, Michael P Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. 2015. Ownership is theft: experiences building an embedded OS in rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. ACM, 21–26.

[26] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. Seattle, WA, 429–444.

[27] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Software fault isolation with API integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 115–128.

[28] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E Nagel. 2015. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *Parallel Processing (ICPP), 2015 44th International Conference on*. IEEE, 739–748.

[29] Andrew C. Myers and Barbara Liskov. 1997. A Decentralized Model for Information Flow Control. In *ACM Symposium on Operating Systems Principles*. Saint Malo, France, 129–142.

[30] Nginx. Nginx: High Performance Load Balancer, Web Server, and Reverse Proxy. https://www.nginx.com/. (????).

[31] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), USENIX OSDI*, Vol. 16.

[32] Zvonimir Rakamarić and Michael Emmi. 2014. SMACK: Decoupling source language details from verifier implementations. In *International Conference on Computer Aided Verification*. Springer, 106–113.

[33] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. 2006. McRT-STM: A High Performance Software Transactional Memory System for a Multi-core Runtime. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06)*. New York, New York, USA, 187–197.

[34] David A. Schmidt. 1998. Data Flow Analysis is Model Checking of Abstract Interpretations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, CA, USA, 38–48.

[35] "servo". Servo web browser engine. http://www.servo.org. (????).

[36] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. 386–400.

[37] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, et al. 2015. Rollback-recovery for middleboxes. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 227–240.

[38] Intel Open Source.org. 2016. Storage Performance Development Kit (SPDK). https://01.org/spdk. (2016).

[39] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An Interaction-based Language and Its Typing System. In *International PARLE Conference on Parallel Architectures and Languages Europe*. 398–413.

[40] Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Compuation* 132, 2 (Feb. 1997), 109–176.

[41] Philip Wadler. 1990. Linear types can change the world!. In *IFIP TC 2 Working Conference on Programming Concepts and Methods*. Sea of Galilee, Israel, 347–359.

[42] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93)*. Asheville, North Carolina, USA, 203–216.

[43] David Walker and Greg Morrisett. 2000. *Alias Types for Recursive Data Structures (Extended Version)*. Technical Report. Ithaca, NY, USA.

[44] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 79–93.

[45] Matteo Zanioli, Pietro Ferrara, and Agostino Cortesi. 2012. SAILS: Static Analysis of Information Leakage with Sample. In *ACM Symposium on Applied Computing*. Trento, Italy, 1308–1313.