Safe Low-Level Code Without Overhead is Practical

Solal Pirelli and George Candea
EPFL

***Abstract*—Developers write low-level systems code in unsafe programming languages due to performance concerns. The lack of safety causes bugs and vulnerabilities that safe languages avoid. We argue that safety without run-time overhead is possible through type invariants that prove the safety of potentially unsafe operations. We empirically show that Rust and C# can be extended with such features to implement safe network device drivers without run-time overhead, and that Ada has these features already.**

***Keywords*—programming languages, safety, performance**

I. INTRODUCTION

Programming languages that provide type and memory safety eliminate entire classes of bugs and security vulnerabilities, yet unsafe languages remain in use even for new projects due to performance concerns. Safe language compilers try to avoid run-time checks if they can prove safety at compile-time but rely on heuristics. Compilers often do not have the information necessary to formally prove the safety of operations at compile-time. We study the necessity of "unsafety". Is there a practical way for languages to provide safety without run-time overhead?

We write network device drivers in three safe languages and a baseline unsafe language to empirically assess performance. Network drivers have nanosecond-level latency requirements, which makes even small inefficiencies matter. We use Ada, Rust, and C# as our safe languages. We choose Ada because of its focus on safety and its sheer number of features. We choose Rust and C# because they are mainstream and represent two major tradeoffs for memory safety. In addition, they both have "unsafe" dialects that we can use to prototype new language features. We implement two sets of drivers, both of which can be used by network software yet are small enough to be ported and audited with reasonable effort.

We advocate for the systematic use of type invariants for safety. Type invariants can give compilers enough information to prove safety while keeping reasoning local and thus scalable. Languages with such invariants enable compilers to trivially prove at compile-time that possibly unsafe operations are safe. For instance, array indexing is safe if the index is guaranteed to be within the array bounds. Some modern languages provide types with safety-related invariants, such as Rust's non-null references, but they do not do so in a systematic fashion for each potentially unsafe operation. Plus, their goal is to help developers reason about correctness, not to prove safety automatically.

Previous work suggests garbage collection in a language is a roadblock to fast network drivers [16], but we argue that this is not the case. Garbage collection is acceptable if the language also provides non-garbage-collected references whose safety is checked at compile-time as an alternative.

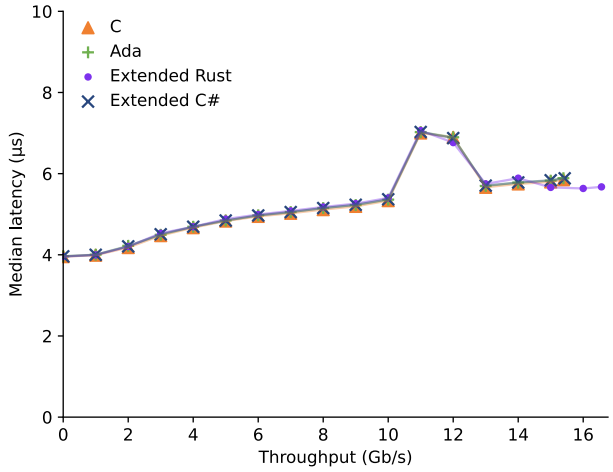

Figure 1. Throughput vs. latency of one of our sets of drivers until the drivers start dropping packets. All drivers are safe and contain no compiler-inserted checks.

Since Rust and C# do not have types with all of the invariants necessary to implement our example drivers, we extend these languages with the features we need. These extensions enable their respective compilers to trivially prove the safety of our network drivers. Surprisingly, Rust's focus on correctness through its ownership semantics comes at the cost of forbidding patterns necessary to safely implement the data structures we need without run-time overhead. As a result, we extend Rust with a new kind of reference that enforces lifetime, which is necessary for safety, but not ownership, which only helps with correctness. We extend C# to add arrays with bounds known at compile-time and enhance its non-garbage-collected reference type. We do not need to extend Ada, as it already has all the type invariants we need.

We empirically evaluate our safe drivers and show that their performance matches that of the same driver written in C, as we preview in Figure 1. Thanks to type invariants, the compilers have enough information to avoid run-time checks and thus our safe drivers reach performance parity with the C baseline. We also observe that the choice of compiler for C has more impact on performance than the choice of compile-time or run-time checks for C#, evidence that the impact of run-time checks on performance is not as strong as one would expect compared to other factors in compilation.

In summary, this paper provides evidence that safe low-level code without overhead is practical using type invariants, though tedious with current languages and compilers because they were not designed for this.

II. PROBLEM STATEMENT

In this paper, "safety" means memory safety and crash freedom: safe code never accesses memory it does not own and never performs invalid operations such as divisions by zero. Such safety prevents vulnerabilities such as Heartbleed [14].

Safe languages such as Java and C# provide safety using both compile-time and run-time checks. For instance, the Java compiler inserts null checks before each dereference. Some checks can be combined or elided, as in the following single-threaded Java example:

```
for (int n = 0; n < array.length; n++) {
    array[n] = 0;
}
```

The compiler is allowed to only check that `array` is not null once before the loop, and to elide the bounds check on the array access entirely since the index is provably in bounds. The HotSpot compiler for Java indeed elides bounds checks [34]. However, these optimizations require heuristics to recognize specific code shapes and do not provide guarantees. The C# compiler currently has open issues regarding elision heuristic failures [9–11], and so does the Rust compiler [37]. Hardware also has heuristics such as branch prediction to alleviate the cost of safety checks, but again without guarantees. We will not discuss hardware heuristics further in this paper.

Developers of performance-sensitive code such as device drivers often use unsafe languages such as C and C++. These languages enable developers to write code without the run-time overhead of safety checks, but in doing so push the burden of ensuring safety onto developers. For data that is internal to the program, such as data structures used to track hardware state, no checks are necessary as long as developers write correct code. For instance, an index into a ring buffer can be used to index into that buffer without checks as long as it starts at a valid value and is reset to 0 once it reaches the end of the ring buffer. This is not feasible in a language such as Java, in which the compiler inserts a safety check for each ring buffer access because it lacks information to prove at compile-time that the index is always updated properly.

Because developers make mistakes, and these have security consequences, there is a large body of work on making C safer. Instead of eliding checks from a safe language, such work adds checks to C. Cyclone [23], Deputy [8], and Checked C [15] all require developers to write annotations such as array sizes to give information to the compiler, which inserts run-time checks if it cannot statically prove safety. Control-C [39] instead limits the expressivity of the language, such as requiring array indexes to occur in specific patterns, and uses a solver to prove safety. Control-C code that cannot convince the solver is disallowed.

Previous work has found a performance penalty from run-time safety checks. Emmerich et al. [16] found slowdowns in the tens of percent for network drivers in safe languages, which they partly attribute to garbage collection. Narayanan et al. [28] wrote a driver in Rust, which does not need garbage collection, but still found overhead compared to C due to run-time checks inserted by the compiler. ASAP [42] reduces the bulk of this overhead by removing the most expensive checks, thus trading safety for performance.

In theory, a sufficiently advanced type system combined with a solver can guarantee safety and even correctness at compile-time. Dependent types, for instance, enable developers to write arbitrarily complex type invariants such as a binary tree being balanced. Using dependent types to avoid safety checks in array indexing was proposed by Xi and Pfenning [44], but requires the use of SMT solving, which is an intractable problem. The Ada 2012 language has dependent types, but checks invariants at run-time [35], thus it supports any invariant developers may write at the cost of run-time checks.

Another theoretical possibility is to manually prove that one's use of unsafe operations is safe. In particular, one can use "unsafe" dialects offered by some languages. In these dialects, operations such as array indexing do not guarantee safety and are thus done without checks, effectively dropping down to the safety level of C for parts of the program. Kiselyov and Shan proposed [25] the use of unsafe code to write "trusted kernels" that wrap general-purpose types into richer types such as "non-empty list" and provide safe operations on these types without run-time checks. Yanovski et al. [45] adapted this idea to Rust using an idea from Beingessner [4], though Rust's type system limits the use of such kernels to specific code patterns. However, these kernels are one-off features that each require manual proofs to ensure they only use unsafe operations in safe ways, which requires expertise and time.

Some programming languages offer practical automated features to avoid safety bugs at compile-time, such as nullable reference types in C# [29]. C# developers can annotate which variables may or may not be null and receive warnings when a possibly null value is assigned to a variable of a non-null type. However, this analysis depends on trusted annotations and is thus unsound [6], because of compatibility concerns with code written before the introduction of this feature.

In this paper, we study whether there is a practical way for programming languages to systematically provide safe low-level operations without run-time overhead. Unlike prior work, we focus on avoiding overhead entirely rather than minimizing it, and we use mainstream programming languages that do not require manual proof effort nor solver-based heuristics.

Another way to view the problem is whether it is possible in practice to give enough information to a compiler so that it can trivially prove at compile-time that each possibly unsafe operation in the code is, in fact, safe. Since we focus on safety rather than correctness, this is a simpler problem than the ones solved by heavyweight techniques such as manual proofs with dependent types. We leave the question of correctness to developers, and only require the language to provide safety without run-time overhead.

If mainstream programing languages can provide safety without run-time overhead, both safety and correctness will benefit. Reasoning about unsafe languages, whether manually or automatically, is considerably harder than reasoning about safe languages. For instance, modeling memory in a way that allows memory-unsafe operations adds complexity. If low-level systems were written in safe languages, not only would entire classes of bugs be eliminated, but verification and bug-finding tools would be easier to develop.

III. CHOICE OF STUDIED LANGUAGES

To study how languages can enable safety without run-time overhead, we start with C as a baseline for performance, and we use three safe languages for comparison.

We choose Ada as the first safe language due to its focus on safety and the sheer number of features it has [19]. Ada is older than most languages still in use and was developed specifically for safety using strong typing such as non-null references, ranged integer types, and arrays with custom bounds. This makes it more likely that Ada may have the features necessary for safety without run-time overhead. Ada does contain unsafe features that can be mixed with safe code without limits, but these are well-known and typically start with the "Unchecked" prefix. For instance, `Unchecked_Deallocation` frees any value without preventing the developer from using the value in source code after it was freed, which is unsafe. Despite its age, there is a modern Ada compiler named GNAT based on GCC, ensuring performance comparisons with modern languages are fair.

To select the remaining safe languages, we use popularity, extensibility, and support for low-level programming as our main criteria. We want languages that are mainstream, can be extended with reasonable effort, and already support most of the operations we need. We want to avoid having to alter the design of languages to the point where they are no longer the same language.

We start from three well-known programming language rankings to estimate popularity: the StackOverflow Developer Survey [38], the RedMonk rankings [40], and the TIOBE index [41]. StackOverflow directly asks developers the languages they use. RedMonk uses GitHub repositories and questions on StackOverflow as proxy metrics for popularity. TIOBE uses search engine results for language names. The methodologies are subject to discussion, but we do not know of a standard way to estimate popularity. We include all languages that reach the top 20 in any of the three rankings, then group them by category as we show in Table 1.

Only two categories from the table are relevant for this study. First, compiled languages with garbage collection, in which we include those that use reference counting such as Swift. Second, "other", which only contains Rust.

Rust enforces memory safety at compile-time through its concepts of ownership, borrowing, and lifetimes. This avoids the need for run-time garbage collection and thus reduces the overhead of memory management, but makes the language harder to use [7]. Rust also enforces "aliasing XOR mutability": a value is, at any given time, accessible by either any number of read-only references or by a single writeable reference, but not both. This prevents correctness bugs such as data races in concurrent code or accidental modification of a data structure while iterating it in single-threaded code. Some data structures such as doubly linked lists are incompatible with Rust's model, thus Rust provides an "unsafe" dialect with raw pointers whose ownership and lifetime are not checked. Rust's unsafe dialect also provides unsafe operations without run-time checks such as unchecked array indexing.

Category	Languages
Compiled, with garbage collection	C#, Dart, Go, Java, Kotlin, Objective-C, Scala, Swift, Visual Basic, VB.NET
Interpreted	JavaScript, Lua, MATLAB, Perl, PHP, PowerShell, Python, R, Ruby, Shell, TypeScript
Markup	CSS
Other	Rust
Query	SQL
Unsafe	Assembly, C, C++, Delphi

Table 1. Programming languages in the top 20 of rankings from the StackOverflow, RedMonk, and TIOBE rankings, sorted by category.

Rust is a natural candidate for this study as it is different from other languages yet relatively mainstream, and its unsafe dialect can be used to write custom types that internally avoid safety checks. In fact, some parts of Rust's standard library contain unsafe code as they cannot be written in safe Rust.

Among the garbage-collected languages we choose C# due to its support for low-level programming and its extensibility thanks to its unsafe dialect. C# provides safety and developer convenience thanks to garbage collection, but also low-level features that remain safe but require more effort to use such as "value types" that do not have object headers and can be passed by reference without garbage collection. The latest C# versions have added more low-level features such as making references to stack-allocated data more flexible [43]. Furthermore, C# has an unsafe dialect that allows the use of pointers, intended for interoperability with unsafe languages. We can use this dialect to prototype language extensions without having to change the compiler or runtime. This also makes C# a good choice given our choice of Rust, since both languages can be extended in a similar fashion.

We explicitly did not choose any interpreted languages, even those that can be compiled to native code such as Python using PyPy. These languages have features with overhead by default, such as Python's unbounded "big integers". Such overhead is not a key issue for languages that are designed to be interpreted, since interpretation already sacrifices speed.

There are plenty of interesting research languages we could have used, but they are typically not designed to be as usable as mainstream languages, and they are not always maintained. For instance, we could have chosen a solver-aided language such as Dafny [26], but it requires manual effort to write proofs in addition to writing code. We could have chosen a research language such as Sing#, a C# extension used for Singularity [20], but it is not maintained thus its compiler may be hard to extend and not include modern optimizations. While Rust is also the subject of formal methods research [24] to avoid bugs in unsafe Rust, developers do not need to know this to write safe Rust.

IV. DOMAIN: NETWORK DEVICE DRIVERS

We choose the domain of network device drivers, specifically user-space ones that bypass the kernel, to empirically evaluate language features in terms of performance. We base our drivers on TinyNF [31], a minimal device driver for the Intel 82599 network card in C. We reuse the TinyNF driver and write another driver using the driver model of DPDK [13], a user-space networking library. DPDK's driver model is more flexible but more complex due to the use of buffer pools. Our choice is based on four reasons.

First, user-space network drivers have real-world use, they are not toy examples. Networking code can use libraries such as DPDK to avoid the overhead of system calls at the cost of exclusive access to a network card. In practice, that network card can be virtualized [30], multiplexing it into virtual cards that applications can have exclusive access to.

Second, network drivers have strict performance requirements. The smallest Ethernet packet, including framing, is 84 bytes, thus handling 10 Gb/s of such packets means each packet must be handled in 67ns. Handling packets in two directions, which is common for network software such as firewalls, halves the time budget. Spreading load across multiple cores can help, but Ethernet speeds are growing as well, currently up to 400 Gb/s [21].

Third, we believe that network drivers are complex enough to be representative of low-level systems code that uses unsafe code for performance but would benefit from safety. The use of buffer pools in the DPDK model means the driver uses a data structure. Data structures are too complex for current automated network software verification techniques, which must therefore assume their correctness [32, 46]. In the case of a buffer pool, the code typically accesses a single element of a buffer array at a time, preventing the use of check elision heuristics that apply to loops over entire arrays.

Fourth, the TinyNF codebase is small enough that we can translate it to safe languages with reasonable human effort. It consists of around a thousand lines of C code, compared to larger codebases such as DPDK which have tens of thousands of lines of code. This small size also means our translations can be audited with reasonable effort.

When we refer to "safe" drivers in this paper, we mean the safety of everything that is not inherently unsafe. The drivers require unsafe operations during initialization. They must query the PCI metadata of the network card to know where the card's registers are mapped in memory, then convert the integer read from this metadata into a pointer to the block of registers. This is done by reading and writing to PCI metadata using port-mapped I/O then asking the OS to map the space of the network card's registers into virtual memory. This is unavoidable given the hardware interface, though this unsafe code can be handled in the OS instead, as in Singularity [20]. The driver must also request "pinned" memory from the OS, i.e., memory whose physical address will not change, then ask the OS for said physical address. This enables such memory to be used by the network card, which uses physical addressing. No OS interactions are needed after initialization, and performance concerns anyway discourage such interactions during packet processing.

V. MEMORY MANAGEMENT

Network drivers use memory pools rather than explicit memory allocation and deallocation for performance reasons, which also helps with safety. Using a memory pool gives developers full control over memory management, including the choice of when to refill the pool if desired, unlike the implicit pools often used by `malloc`/`free` implementations. Using a pool also turns "use after free" bugs into correctness issues. For instance, if a buffer is simultaneously used by the network card to receive data and by the application using the driver to process a packet, this is a correctness bug, but the code is safe. We use "safety" here to mean that programs cannot interfere with each other or with the OS. A safe program may still have security issues among its own tenants, such as a firewall accidentally sharing state between connections.

Our safe driver implementations need to avoid performance overheads in the implementation and use of a memory pool. Overheads in the default memory allocation and deallocation operations, such as in garbage collection, only matter if they prevent the implementation of an overhead-free memory pool. Overheads due to run-time checks in operations such as array indexing in pool implementations are the same fundamental problem as checks in driver implementations. We can avoid such overheads in the same way as we do in the drivers, as we explain below.

Ada and Rust both support memory management without run-time overhead through language features, allowing for a memory pool that performs the same as in C. Ada does have run-time checks for memory operations that touch different pools, as support for storage pools is built into the language, but these checks can be avoided by using the same pool in the entire codebase. Rust, by design, uses only compile-time checks for memory safety.

This leaves C#, whose garbage collection adds run-time overhead to memory operations even when a garbage collection is not in progress. This is a tradeoff for overall performance: in theory, the collector could go through the entire heap every time, but this is too slow on large heaps. Instead, the collector tracks changes to references with help from the compiler. When a reference to an object is modified, the compiler adds code that sets a specific bit in a table used by the collector, ensuring that the collector knows exactly where to look. In theory, one could disable garbage collection entirely, but this is not practical as it would prevent memory deallocation to preserve memory safety.

This leaves one option for garbage-collected languages: provide some kind of reference that is not garbage-collected and instead follows rules more akin to Ada and Rust. This is a form of arena-based memory management, in which allocations within an "arena" are handled specially and objects outside of the arena cannot refer to memory inside the arena. C# has a basic form of arena-based memory management: developers can declare types that can only live on the stack, and a special kind of reference that can only point to these types and that can itself not be stored on the heap. This is an intermediate design point that is as safe and fast as Rust but not as expressive, in exchange for being simpler to use.

VI. SAFETY WITH TYPE INVARIANTS

As we defined in §II, the problem is whether developers can give enough information to a compiler to prove at compile-time that all possibly unsafe operations are safe.

We propose the use of *type invariants* as a solution to this problem, i.e., predicates that hold on all instances of a type. Type invariants are already used in practice, but we propose that languages should systematically provide them for each possibly unsafe operation. Consider the following C code:

```
int* value;
bool set_value(int* v) {
  if (v == NULL) { return false; }
  value = v;
  return true;
}
```

The `set_value` function guarantees that value is non-null. Thus, a developer can use `value` once `set_value` has been called without having to write a safety check. Consider the following C# code, which looks equivalent:

```
class Example {
  int[] value;
  Example(int[] v) {
    if (v == null) { throw new ...; }
    value = v;
  }
}
```

The same invariant holds: `value` is not null once the object is initialized. However, this information is not available to the compiler outside of the `Example` constructor, as it is not a type invariant of `int[]`. The compiler can only analyze signatures rather than implementations, since there are too many paths within implementations [5]. Thus, it must insert a null check when `value` is used, as the signature lacks information to prove `value` is not null. C# developers can optionally annotate their code with nullability information, but the compiler cannot trust that the annotations are correct because they can describe arbitrarily complex conditions and thus must insert checks anyway.

Rust, however, does have an invariant for "non-null", which is in fact the default:

```
// Lifetime annotations omitted for brevity
struct Example {
  pub value: &[u32]
}
```

The member `value` in this struct is never null, thus the compiler does not have to insert null checks when it is used. Instead, the compiler enforces that any instance of `Example` is created with a non-null `value`. If a developer wants a possibly absent array, they can use Rust's `Option` type to wrap the array type, at which point the compiler requires a check before every use to ensure the `Option` indeed contains an array.

Rust enables safe code without null checks, but null checks are not the only kind of checks that are typically necessary in safe languages. Going back to the Rust struct above, indexing the array forces the compiler to insert a bounds check, unless the compiler can prove that the index is in bounds, for instance in a loop.

Type invariants can help avoid bounds checks just as they do with null checks. Consider the following Ada declarations:

```
-- Ranges are inclusive in Ada,
-- i.e., 0 <= n <= 9
type Small is range 0 .. 9;
type SmallArray is array(Small) of Integer;
```

The first declaration is for a ranged integer, and the second is for an array indexed by that range. Because this information is contained within the type's invariant, an Ada compiler can omit bounds checks when a `SmallArray` is indexed by a `Small`, as any value of type `Small` must be in bounds. The developer or the compiler must instead insert checks when converting from an integer with a different range, since the integer may not be in the target range's bounds.

Safety checks when converting a value of a general-purpose type to a type with an invariant are not overhead compared to C, because when writing C, developers must write these checks manually. For instance, checking that some user input is smaller than the size of an array is necessary if the input is used to index that array later. But in languages without type invariants, this check is only recorded in the implementation of some function and in the developer's memory. With type invariants, compilers can use signatures to know that a check was performed and avoid generating pointless run-time checks.

Programming languages can thus enable safe code without overhead as long as they only permit unsafe operations on types with invariants that trivially prove safety. Converting a piece of untrusted data to a type with the right invariant is only needed once, and it is not overhead since unsafe code must also convert and potentially reject the input. Once a value is converted to a type with an invariant, some operations on that value return a value with that same invariant, while others need a conversion. For instance, dividing a value of type "integer modulo N" by 2 returns a value of the same type, since the resulting value must mathematically be below N. However, incrementing that integer returns a value with a less strict type invariant, since the value can exceed N. Converting the result back to its original type code can be implicit given hardware instructions that match. For instance, incrementing an "integer modulo 256" can be done with a "byte increment" instruction.

Type invariants also enable compilers of safe languages to avoid memory overheads in the same way a programmer would in an unsafe language. Consider again Rust's `Option` type, which represents "a value, or nothing". Since Rust references cannot be null, per their type invariant, the compiler can use the memory representation of "null", e.g., all-zeroes, to represent an empty `Option` of a reference type, instead of using a separate Boolean value to track whether the option contains a value. Rust similarly has "non-zero" integer types with an invariant enabling the compiler to optimize the representation of `Option`.

We present below a systematic overview of type invariants that modern languages need to provide safety without run-time overhead. This list matches the potentially unsafe operations exposed by modern programming languages for low-level code: using references, dividing integers, and indexing arrays. It may not be exhaustive for future languages that could add more potentially unsafe operations.

Valid references. To avoid overhead, memory safety must be a type invariant, i.e., references that always point to valid memory without the need for run-time bookkeeping. This means references cannot be null and cannot require extra work such as garbage collection to ensure memory is properly freed.

One way to implement this is through memory arenas: by allocating a group of objects together, each object can refer to any other object in the same arena without overhead, and the arena is freed as a whole, at which point accessing any of its contents is a compile-time error.

C# implements a limited form of memory arenas: it supports special references that can only point to types on the stack and cannot be stored on the heap, thus there is one arena for the stack, with compile-time checks and no run-time overhead, and one arena for the heap, without compile-time checks but with the run-time overhead of garbage collection.

Rust implements valid references through its more complex ownership model, which enables more scenarios than C#'s stack-only references but requires more developer effort. The lifetime of Rust references must often be specified explicitly.

Ranged integers. To avoid overhead, safe languages must provide integer bounds as type invariants, including bounds not known at compile-time such as the number of network cards. This enables compilers to prove the absence of division by zero.

Most languages already provide a few specific bounds that correspond to machine types, such as C's `uint16_t` for integers between 0 inclusive and 2^{16} exclusive. This enables compilers to translate operations in these types to the equivalent machine operations, such as addition of 16-bit integers. However, some high-level languages have no ranged integers at all. In Python, for instance, all integers are mathematical integers of infinite range, thus all integer operations incur the overhead of general-purpose mathematical integer arithmetic. Even with a fast path for "sufficiently small" integers, the code has the overhead of checking if the fast path applies.

Ranged arrays. To avoid overhead, safe languages must provide arrays that encode their length, i.e., the range of valid indices, in their type. In combination with ranged integers, this enables a compiler to prove that an array access is valid at compile-time instead of requiring run-time checks.

Array ranges must be allowed to have bounds not known at compile-time, as with integer ranges. For instance, a variable must be allowed to contain the index of the network card that was last queried for new packets and be used to index an array of network cards, even though the number of network cards is machine-dependent and unknown at compile-time. Rust in particular does not meet this criterion: its array types can include their length, but only if the length is a compile-time constant.

All three of our proposed invariants enable developers to give information to the compiler explicitly and formally instead of keeping it in their head. Developers in C must keep track of which references are guaranteed to be valid and which are not, of what range each integer variable has, and of what range each block of allocated memory can be indexed with. For ranges that are not compile-time constants, this is typically done by making the lower bound 0 and storing the upper bound in a variable, as a compiler would for a ranged array or integer type.

```
pub struct LPtr<'a, T> {
  ptr: NonNull<T>,
  _lifetime: PhantomData<&'a mut T>,
}
impl<'a, T> LPtr<'a, T> {
  pub fn new(src: &'a mut T) -> LPtr<'a, T>;
  pub fn read_volatile(&self) -> T;
  pub fn write_volatile(&self, value: T);
  pub fn map<U, F>(&self, f: F) -> U
    where F: FnOnce(&mut T) -> U;
}
```

Figure 2. Extract from our custom reference type that enforces lifetime but not ownership. Some methods and annotations omitted for readability.

VII. PROTOTYPE

We write safe drivers, as defined in §IV, in Ada, Rust, and C#, and a baseline unsafe driver in C. To do so, we extend Rust and C# to add limited forms of the type invariants we described.

In Ada, the only issue we have is that ranged array types by default carry both upper and lower bounds, even if the latter is 0, requiring extra memory. We use a workaround suggested by the AdaCore developers [1] to store only the upper bound.

In Rust, we implement a limited form of ranged arrays by using existing compiler support for eliding checks if an access is performed with an integer of a small enough type, such as indexing an array with 256 elements using an 8-bit unsigned integer. This causes small memory overheads for arrays that do not need 256 elements but must have them to benefit from bounds check elision, which we find acceptable for a prototype.

Interestingly, Rust's "aliasing XOR mutability" model is too restrictive for device drivers. Some driver instances must share state, such as a buffer pool used by both receive and transmit queues. Both queues must be able to mutate the pool: receive queues take buffers and transmit queues give buffers back. However, in safe Rust, the queues cannot simultaneously have a mutable reference to the pool. Furthermore, drivers need to use volatile reads and writes for memory-mapped I/O, since the network card can change its data at any time. The card logically has a mutable reference to its own data. But in Rust's model, if the card can mutate itself, then nobody else should be allowed to even access it. The solution used by the Rust version of the Ixy [16] network driver is to use unsafe Rust code, forgoing safety guarantees. Instead, we write our own type of reference that internally uses unsafe code but exposes a safe interface, which we show in Figure 2. Our custom `LPtr` type enforces lifetime, and thus memory safety, but not ownership, leaving correctness to developers. It has a field of type `PhantomData`, a special Rust type that evaporates during compilation and only serves as a marker for data lifetime in niche scenarios such as implementing a reference. In addition, we implement an array type with lifetime but not ownership, matching our `LPtr` type.

In C#, we use the same bounds check elision trick as in Rust, with the same overhead, but we first extend the language with ranged arrays. The C# maintainers have prototyped such arrays already [12], but they are not yet part of the language. We also add support for arrays of stack-only references.

VIII. EVALUATION

In this evaluation, we show that (1) our drivers are safe, (2) they have no run-time performance overhead compared to the unsafe baseline, and (3) our approach leads to safe code that is easy to read but not easy to write, as current compilers are not designed for this and thus make the task harder than it needs to be.

As we stated in §IV, we wrote safe drivers in Rust, C#, Ada, and unsafe baselines in C, using two models: a "restricted" and a "flexible" one. Our C baselines perform no run-time safety checks. We use Clang 13 for C, Rust 1.58, .NET 6, and GNAT 11 for Ada. We intended to use GCC for C, but it produces a slower binary than Clang, as we describe later. For C# we use ahead-of-time compilation, which produces the same code as the default just-in-time compilation, so that we can inspect the assembly code that we run.

Are our drivers safe? Our safe drivers only use unsafe code during initialization for the steps that are inherently unsafe as we describe in §IV, and for the implementation of our Rust and C# extensions. Our extensions are about 200 lines each of Rust and C# that we believe but do not formally prove are safe. Thus, our drivers are safe assuming the correctness of the languages themselves as well as our extensions.

Do our safe drivers impose run-time overhead? We benchmark our safe drivers to ensure their performance matches our baseline in practice, since checking the exact equivalence of assembly code is not practically feasible. We use two machines in a setup based on RFC 2544 [36]: a "device under test" runs the driver under test and a "tester" runs the MoonGen packet generator [17]. Both machines run Ubuntu 18.04 on two Intel Xeon E5-2667 v2 CPUs with power-saving features disabled and have two Intel 82599ES NICs. We use only one Ethernet port per card to ensure PCIe bandwidth is not a bottleneck. We measure throughput using minimally sized packets: 64 bytes of content plus 20 bytes of Ethernet framing. We transmit such packets for 30 seconds at a configurable rate. Our drivers are single-threaded. We set the CPU frequency to 1.5 GHz instead of the default 3.6 GHz of this CPU model, because otherwise our drivers saturate the links and exhibit identical performance. We write "forwarder" programs on top of the drivers that modify the source and destination MAC addresses of each packet and forward it to the opposite card. We measure the highest throughput at which the drivers do not drop packets, then we measure the latency of packets in increments of 1 Gb/s from 0 to the maximal loss-free throughput. This is the same benchmark used to evaluate the original TinyNF driver [31].

We first benchmark the drivers for the "restricted" model, in which a single driver instance combines reception and transmission to share data structures and thus minimize overhead. This model can only be used by network software that handles packets one by one without reordering them. We show the results in Figure 3, which we already previewed at the beginning of this paper. The results are nearly identical for the different drivers, except that the Rust version sustains more throughput. These results are consistent across different benchmark runs. The latency bump around 11 Gb/s is odd, but the TinyNF paper already reported it [31] even when using DPDK and Ixy, which use entirely different codebases. It is likely a hardware issue.

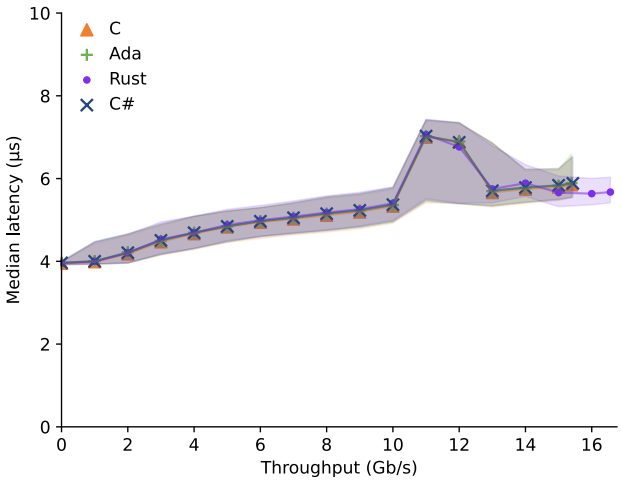

Figure 3. Throughput vs. latency until the drivers start dropping packets for our drivers using the "restricted" model, with a shaded 5-95% ranges for latencies.

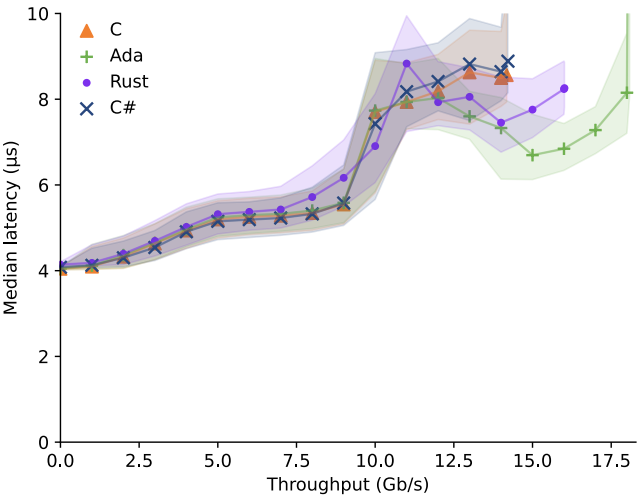

Figure 4. Throughput vs. latency until the drivers start dropping packets for our drivers using the "flexible" model, with a shaded 5-95% ranges for latencies.

We then benchmark the drivers for the "flexible" model, in which reception and transmission use a shared buffer pool. This model supports all network software, including those that must reorder packets. We show the results in Figure 4. C# is still close to C, with slightly higher latency at the highest load. Rust keeps its advantage compared to C, though it has higher latencies at lower loads. Surprisingly, the Ada version can sustain higher throughput not only compared to the other drivers of the same model but even to the drivers of the restricted model, albeit with higher latency than the restricted model. We double-checked the Ada code to ensure it performs the same tasks in the same order including volatile reads and writes. The only explanation we can find is that, to remove bounds checks from the Ada code, we use more specific types than in other languages. In particular, we use integers bounded to the batch size for reception and to the number of received packets for transmission. The Ada compiler may produce better code when using Ada's bounded integers, since they provide additional information.

We investigate further by writing drivers using the "restricted" model but with a static output count. That is, instead of determining the number of network cards at run-time, this number is known at compile-time using the pre-processor in C, constant generic parameters in Rust, and generic values in Ada. C# has no such feature. This requires re-compiling the source code for each deployment, but it could be practical for network software that intrinsically has a fixed output count. For instance, a firewall could have "internal" and "external" virtual devices and be chained with a router whose number of outputs depends on the actual number of physical devices. In theory, using static output counts gives compilers more room for optimizations, such as unrolling loops. We show the results in Figure 5. Ada reaches the same maximum throughput as the version with the flexible model but with a lower latency, which is expected given this restricted model. Rust performs almost identically to Ada, within the noise of small variations across experiments. The C version barely improves with a static output count. Since we used language features in Rust and Ada for this experiment but had to use the more general pre-processor in C, this provides more evidence that code using specific language features may enable compilers to optimize better.

To measure the impact of run-time checks on performance, we write a C# driver without our C# extensions. The compiled assembly code of this driver thus includes run-time checks, unlike other drivers we wrote. We compare it to our C driver using different compilers, to use the performance difference between compilers of the same language as a baseline. This leads to unexpected results, which we show in Figure 6. First, using GCC to compile our C driver leads to a larger performance penalty than we expected. We confirm by inspecting the assembly code that GCC spills many more values to the stack than Clang does. Second, the impact of run-time checks in the C# version is lower than the impact of using a different C compiler. In fact, the C# version with run-time checks has similar performance to the C version, though the former's latency spikes near its breaking point in terms of throughput. These results suggest that run-time safety checks may have less impact on performance than the specific optimization choices made by compilers.

In addition to benchmarks, we compare the size of our drivers in terms of lines of code and resulting assembly instructions. While such comparisons are not precise since individual lines of code and assembly instructions vary in complexity, it gives an idea of how close the sizes of our drivers are. We present the results in Table 2. All driver sources have roughly the same size, as expected since most lines read and write to the same network card registers no matter the language. However, the assembly code generated by the Rust compiler is larger than that of the other three. We manually inspect the assembly code and find that the Rust compiler unrolls more loops. We manually confirm that forcing C compilers to do the same does not change performance, though Rust's choice to unroll may be due to the specifics of the assembly it produces rather than a general optimization. This may be why the Rust restricted driver performs better than the ones in other languages.

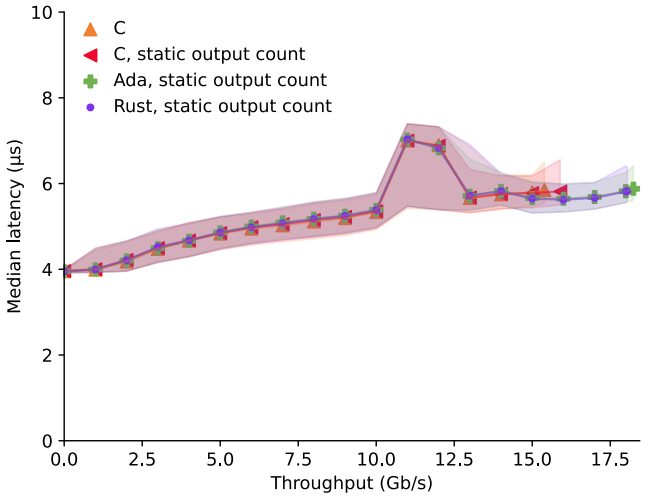

Figure 5. Version of Figure 3 using a static output count.

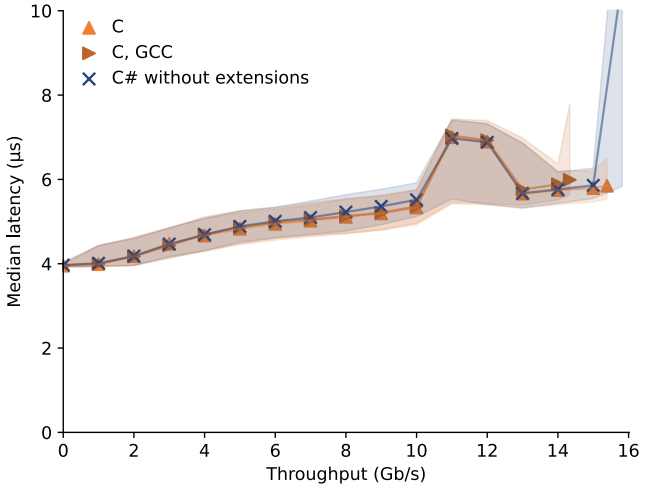

Figure 6. Version of Figure 3 using GCC for C and without our C# extensions.

Language	Lines of code	x86 instrs in main loop	
		Restricted	Flexible
C (unsafe)	1256	267	418
Rust	1114	586	791
C#	1277	233	439
Ada	1261	229	375

Table 2. Code metrics for our drivers. The two driver models share initialization code thus we count the combined lines of code. The main loop includes only packet processing code.

We manually inspect the assembly code to see whether the Ada, Rust, and C# compilers insert checks. These checks call specific functions when they fail, thus we only need to look for calls. We confirm that there are none. We also confirm that the source code does not contain more checks than the C version. The size of the codebases makes such an audit tractable.

Does our approach lead to maintainable code? We evaluate the maintainability of our drivers qualitatively by inspecting their source code. To keep comparisons fair, we do not consider our language extensions, as they are only prototypes in unsafe code and thus require worse syntax to use than real extensions would. Overall, the answer is mixed: the code of our drivers is as easy to read as other code in the respective languages but writing and evolving it is difficult due to the lack of language and compiler support.

The source code of the final version of our drivers looks like normal code in their respective languages. Besides the use of our language extensions, there is little that would tip a reader that we carefully wrote the code to avoid overhead. It is possible to avoid overhead without having to write unusual or contorted code, but this does not mean it is easy.

Writing our drivers was difficult because current compilers were not designed to statically prove safety. Small changes in the source code cause large changes in the compiled assembly code. For instance, the GNAT Ada compiler accepts code that looks safe to developers but in fact requires safety checks at run-time due to language semantics and inserts checks without a warning nor any other means of "debugging" this behavior.

The process of writing safe code without overhead today, even in Ada, which has the necessary features already, is similar in theory to writing manual proofs of correctness but worse in practice because of the lack of tooling. Proofs typically read well once they are finished, yet slight changes in which lemmas are used and in which order can break the proof due to the proof checker being heuristic-based. Similarly, the code of our drivers reads well yet slight tweaks cause run-time checks as compilers can no longer prove the checks are unnecessary. However, whereas proof checkers enable developers to write intermediary assertions and provide information about why the proof process failed, compilers provide no way to access their internal logic. Without such information, writing safe code without overhead is tedious and slow. Developers must guess what source code might cause the compiler's heuristics to output the desired assembly code and check that output after every change until they find a way to convince the compiler to not insert checks.

This situation is made worse by the impact on performance of minor differences in assembly code. We found such minor differences that caused performance regressions on the order of 10%. Thus, one must first spend time convincing the compiler to not insert run-time checks, then benchmark the code. If the code is not fast enough, one must convince the compiler again, this time using different code with the same semantics. Editing the assembly by hand is possible but unsafe and error-prone.

The safety of Ada, Rust, and C# did help us when writing drivers. For instance, we model network card registers as indexes in a buffer. We made copy-paste errors while translating C constants for register indexes, leaving some of them orders-of-magnitude too large. In C, such code silently does not work, as the OS maps more memory than we need. In safe languages, this fails with a descriptive error message. The type invariants we used also caught programming errors such as forgetting to initialize variables, whereas in C the lack of such invariants means that these errors cause compiler warnings at best.

IX. DISCUSSION

We discuss why unsafety is neither necessary nor sufficient to avoid checks, how run-time checks impact performance, how our recommendations could be implemented in practice, the significance of our work for software engineering, and the limitations of our proposal.

Performance requires information, not unsafety. Our drivers are fast because we give compilers enough information to optimize them, using our language extensions when needed. Unsafety can be an escape hatch to implement extensions, but it is neither necessary nor even sufficient. Unsafe languages can also have semantics unfriendly to optimizers, causing overhead and noise. For instance, the default aliasing model of C and C++ pointers forces compilers to assume that any pointer to `char` could point to any variable. Thus, writing to a vector of 8-bit `char`s in a loop is slower than the same operation to a vector of 32-bit integers because the compiler must assume the vector length might be modified by any write to a `char` pointer [22]. The C99 standard improved this situation with the `restrict` keyword to disable this aliasing model, and we use it in our driver. Even with C's unsafety, there was no way to give the compiler aliasing information before the `restrict` keyword, thus imposing a performance penalty even on unsafe code.

Thankfully, while global information is required to prove correctness, only local information is required to prove safety in practice. Type invariants are enough to show that individual operations are safe, and compilers can use them automatically because they only need to prove that a specific type matches a specific operation rather than having to combine invariants into higher-level proofs as for correctness.

The importance of local reasoning is also why the presence of garbage collection is not an issue on its own. One can write safe low-level modules without overhead in a garbage-collected language as long as the language also has a form of memory management without overhead. which does not need to be as convenient as garbage collection. For instance, Frampton et al. [18] extended Java for "high-level low-level programming", enabling developers to use Java for a memory manager as long as they only use low-level operations. High-level modules can use garbage collection if its overhead is negligible compared to other operations such as network requests.

Run-time checks may not be as bad as one expects since performance at the nanosecond scale already varies depending on minor changes in compiler heuristics. The special C# driver without language extensions we used for Figure 6 is only barely worse than the C# driver with our language extensions, and both are better than our C driver compiled with GCC instead of Clang. Inspecting the assembly code does not reveal obvious culprits, as the "CPU-friendliness" of assembly code is hard to evaluate. This is consistent with results from Popescu et al. [33], who insert bounds checks in Rust code that developers manually elided with unsafe code. Adding the checks back sometimes improves performance, for instance because a function that is not profitable to inline in practice is inlined by the compiler when it has no bounds checks due to being small enough. Safety checks also have an effect on code alignment, which is important in practice especially for loops [27].

Tooling support is necessary for performance. We need an evolution, not a revolution. Programming languages already have types with invariants built in, and compilers already try to automatically prove that run-time checks are unnecessary. To enable safe code without run-time overhead, languages should take a more systematic approach to the types they provide given the possibly unsafe operations they provide, and compilers should surface the reasoning they perform internally. The C# language and compiler already have a limited form of this for the C# "nullable reference types" feature: the language exposes both nullable and non-nullable reference types, and the compiler warns when it cannot prove that the target of a dereference is non-nullable. The developer then has two options: they can attempt to modify their code to prove non-nullability, or they can use the "null-forgiving" operator "`!`" to insert a run-time check instead. In practice, even non-nullable dereferences need a runtime check as the compiler's analysis is unsound for compatibility purposes, but this does not need to be the case in a new language.

Language support could also make writing code that uses precise types easier, such as Scala 3's path-dependent types [3] that let developers define the return type of a function in terms of its parameters. Without such support, writing functions such as "index of a character in a string" is cumbersome as it requires explicitly defining the type and bounds of the string.

The benefits of safety without run-time overhead go beyond pure performance improvements for safe code. The need for performance is a key reason unsafe languages are still used for new projects, rather than being relegated to legacy codebases. Past safe languages such as Pascal or Objective-C have all but disappeared in favor of modern safe languages such as Java and Swift, but the unsafe C and C++ are still in use.

Giving software engineers the performance benefits of C with the safety benefits of modern high-level languages would free resources used to maintain tooling for unsafe languages. In particular, the need to provide safety for modern codebases written in unsafe languages requires engineering and research time. Handling C-style unsafe semantics in a verification tool requires considerable effort compared to handling languages that provide memory and type safety. Even if an analyzer for C code requires memory safety, it must properly detect and report code that is not memory safe to have good usability, since a programmer might accidentally write unsafe code.

We do not mean that languages such as Cyclone [23] and Checked C [15] are not useful compared to our proposal. When the risks of existing legacy code are high enough, annotating and adding run-time checks to existing C code is valuable. They also enable developers to use their existing C expertise.

The limitations of our proposal to provide type invariants to enable developers to write safe code without overhead are related to developer effort. Software engineers can write safe code in unsafe languages without explicitly proving why their code is safe. With our proposal, this is not possible. Engineers must either explicitly use the right types for their code and structure their code so the invariants hold or accept a run-time check and the corresponding performance penalty.

X. THREATS TO VALIDITY

This study is limited by scale. The effort of writing safe drivers without run-time overhead scales exponentially with the number of languages and hardware platforms, particularly due to the time it takes to implement a change in a safe language while also convincing the compiler to not insert checks.

Internal validity. We used one kind of network card on one kind of server. Our results may not reproduce exactly on other kinds of hardware since performance at the nanosecond scale is particularly affected by minor variations such as cache size. Faster network speeds, such as 400 Gb/s Ethernet, may bring new challenges. The specific compiler versions we used may also affect results, as we have already noticed a performance difference between GCC and Clang for the baseline C driver.

Construct validity. We inspected the compiled code of our drivers to confirm the lack of run-time checks and we measured their performance empirically, but these are only proxy metrics for equivalence. Ideally, we would prove that the compiled code of each safe driver is exactly equivalent to that of the baseline, but this is not currently feasible given the complexity of x86 assembly code, the complexity of automated verification, and the lack of a definition for what equivalence even means, given that not all side effects are relevant.

We assumed that the safe languages we use are safe. This may not be true, since neither the language specifications nor the compilers we use are formally proven to be correct. Ada's specification, in particular, is so complex that the authors of GNAT do not believe they fully understand the rules around reference safety [2].

External validity. We chose network drivers for reasons we outlined in §IV, but they may not generalize to all kinds of low-level systems. In particular, we did not need high-level safety features such as Rust's ownership system that prevents data races as we do not use multi-threading. In fact, in this study, Rust's ownership system was more of a hindrance than a help.

XI. CONCLUSION

We provide evidence that safety without run-time overhead is practical. If languages provide types with invariants that match the requirements of potentially unsafe operations, such as ranged integers and arrays, compilers can trivially prove safety at compile-time. Checking that untrusted data satisfies the invariant is necessary, but this is also the case in unsafe code.

We build network card drivers in Ada, Rust, and C#, as well as a baseline in C. To do so, we extend Rust and C# with new features that enable the driver code to not need any run-time checks. Empirical performance evaluation at the nanosecond scale reveals that the performance is indeed on par with C as we expect. Interestingly, Ada already has the necessary features for safety without run-time overhead.

We hope these results encourage future work on safe low-level systems code without overhead.

XII. DATA AVAILABILITY

All code and data for this paper are publicly available at https://github.com/dslab-epfl/tinynf.

REFERENCES

[1] Ada RFC: Lower Bound Constraint: *https://github.com/AdaCore/ada-spark-rfcs/blob/master/considered/rfc-lower-bound.rst*.

[2] Ada RFC: Simpler accessibility rules: *https://github.com/AdaCore/ada-spark-rfcs/pull/47*.

[3] Amin, N., Rompf, T. and Odersky, M. 2014. Foundations of Path-Dependent Types. *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (New York, NY, USA, 2014), 233–249.

[4] Beingessner, A. *You can't spell trust without Rust*. Carleton University.

[5] Boonstoppel, P., Cadar, C. and Engler, D. 2008. RWset: Attacking Path Explosion in Constraint-Based Test Generation. *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008), 351–366.

[6] C# Reserved attributes: Nullable static analysis: 2021. *https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/attributes/nullable-analysis*.

[7] Coblenz, M., Mazurek, M.L. and Hicks, M. 2022. Garbage Collection Makes Rust Easier to Use: A Randomized Controlled Trial of the Bronze Garbage Collector. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)* (2022), 1021–1032.

[8] Condit, J., Harren, M., Anderson, Z., Gay, D. and Necula, G.C. 2007. Dependent Types for Low-Level Programming. *Proceedings of the 16th European Symposium on Programming* (Berlin, Heidelberg, 2007), 520–535.

[9] dotnet/runtime issue #9422: Loop condition doesn't elide bounds check: 2021. *https://github.com/dotnet/runtime/issues/9422*.

[10] dotnet/runtime issue #11359: Regressions in span indexer performance: *https://github.com/dotnet/runtime/issues/11359*.

[11] dotnet/runtime issue #44415: Bounds checks are no longer elided when using nint for indexing: *https://github.com/dotnet/runtime/issues/44415*.

[12] dotnet/runtime pull request #60519: ValueArray: *https://github.com/dotnet/runtime/pull/60519*.

[13] DPDK: 2021. *https://www.dpdk.org/*.

[14] Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M. and Halderman, J.A. 2014. The Matter of Heartbleed. *Proceedings of the 2014 Conference on Internet Measurement Conference* (New York, NY, USA, 2014), 475–488.

[15] Elliott, A.S., Ruef, A., Hicks, M. and Tarditi, D. 2018. Checked C: Making C Safe by Extension. *2018 IEEE Cybersecurity Development (SecDev)* (2018), 53–60.

[16] Emmerich, P., Ellmann, S., Bonk, F., Egger, A., Sánchez-Torija, E.G., Günzel, T., Luzio, S.D., Obada, A., Stadlmeier, M., Voit, S. and Carle, G. 2019. The Case for Writing Network Drivers in High-Level Programming Languages. *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2019)* (Sep. 2019).

[17] Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F. and Carle, G. 2015. MoonGen: A Scriptable High-Speed Packet Generator. *Proceedings of the 2015 Internet Measurement Conference* (New York, NY, USA, 2015), 275–287.

[18] Frampton, D., Blackburn, S.M., Cheng, P., Garner, R.J., Grove, D., Moss, J.E.B. and Salishev, S.I. 2009. Demystifying Magic: High-Level Low-Level Programming. *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2009), 81–90.

[19] Hoare, C.A.R. 2007. The Emperor's Old Clothes. *ACM Turing Award Lectures*. Association for Computing Machinery. 1980.

[20] Hunt, G.C. and Larus, J.R. 2007. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.* 41, 2 (Apr. 2007), 37–49. DOI:https://doi.org/10.1145/1243418.1243424.

[21] IEEE Standards Association IEEE 802.3bs-2017.

[22] Incrementing Vectors: *https://travis-downs.github.io/blog/2019/08/26/vector-inc.html*.

[23] Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J. and Wang, Y. 2002. Cyclone: A Safe Dialect of C. *2002 USENIX Annual Technical Conference (USENIX ATC 02)* (Monterey, CA, Jun. 2002).

[24] Jung, R., Jourdan, J.-H., Krebbers, R. and Dreyer, D. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL (Dec. 2017). DOI:https://doi.org/10.1145/3158154.

[25] Kiselyov, O. and Shan, C. 2007. Lightweight Static Capabilities. *Electron. Notes Theor. Comput. Sci.* 174, 7 (Jun. 2007), 79–104. DOI:https://doi.org/10.1016/j.entcs.2006.10.039.

[26] Leino, K.R.M. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Berlin, Heidelberg, 2010), 348–370.

[27] Loop Alignment in .NET 6: *https://devblogs.microsoft.com/dotnet/loop-alignment-in-net-6/*.

[28] Narayanan, V., Huang, T., Detweiler, D., Appel, D., Li, Z., Zellweger, G. and Burtsev, A. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), 21–39.

[29] Nullable reference types - C# guide: *https://docs.microsoft.com/en-us/dotnet/csharp/nullable-references*.

[30] Peter, S., Li, J., Zhang, I., Ports, D.R.K., Woos, D., Krishnamurthy, A., Anderson, T. and Roscoe, T. 2015. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.* 33, 4 (Nov. 2015). DOI:https://doi.org/10.1145/2812806.

[31] Pirelli, S. and Candea, G. 2020. A Simpler and Faster NIC Driver Model for Network Functions. *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), 225–241.

[32] Pirelli, S., Valentukonytė, A., Argyraki, K. and Candea, G. 2022. Automated Verification of Network Function Binaries. *NSDI* (2022).

[33] Popescu, N., Xu, Z., Apostolakis, S., August, D.I. and Levy, A. 2021. Safer at Any Speed: Automatic Context-Aware Safety Enhancement for Rust. *Proc. ACM Program. Lang.* 5, OOPSLA (Oct. 2021). DOI:https://doi.org/10.1145/3485480.

[34] Range Check Elimination - OpenJDK HotSpot Wiki: 2021. *https://wiki.openjdk.java.net/display/HotSpot/RangeCheckElimination*.

[35] Rationale for Ada 2012 - Subtype predicates: *http://www.ada-auth.org/standards/12rat/html/Rat12-2-5.html*.

[36] RFC 2544 - Benchmarking Methodology for Network Interconnect Devices: 1999. *https://www.ietf.org/rfc/rfc2544.txt*.

[37] rust-lang/rust issue #81253: Array bound tests with for loop that get removed with while loops: *https://github.com/rust-lang/rust/issues/81253*.

[38] StackOverflow 2022 Developer Survey: *https://survey.stackoverflow.co/2022/*.

[39] Sumant Kowshik, D.D. and Adve, V. 2002. Ensuring Code Safety Without Runtime Checks for Real-Time Control Systems. *Proc. Int'l Conf. on Compilers Architecture and Synthesis for Embedded Systems, 2002* (Grenoble, France, Oct. 2002).

[40] The RedMonk Programming Languages Rankings: January 2022: *https://redmonk.com/sogrady/2022/03/28/language-rankings-1-22/*.

[41] TIOBE Index: *https://www.tiobe.com/tiobe-index/*.

[42] Wagner, J., Kuznetsov, V., Candea, G. and Kinder, J. 2015. High System-Code Security with Low Overhead. *2015 IEEE Symposium on Security and Privacy* (2015), 866–879.

[43] Write safe and efficient C# code: *https://docs.microsoft.com/en-us/dotnet/csharp/write-safe-efficient-code*.

[44] Xi, H. and Pfenning, F. 1998. Eliminating Array Bound Checking through Dependent Types. *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1998), 249–257.

[45] Yanovski, J., Dang, H.-H., Jung, R. and Dreyer, D. 2021. GhostCell: Separating Permissions from Data in Rust. *Proc. ACM Program. Lang.* 5, ICFP (Aug. 2021). DOI:https://doi.org/10.1145/3473597.

[46] Zhang, K., Zhuo, D., Akella, A., Krishnamurthy, A. and Wang, X. 2020. Automated Verification of Customizable Middlebox Properties with Gravel. *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, Feb. 2020), 221–239.