Verifying Device Drivers with Pancake

Junming Zhao[1], Alessandro Legnani[2,3], Tiana Tsang Ung[1], H. Truong[1], Tsun Wang Sau[1], Miki Tanaka[1], Johannes Åman Pohjola[4,5,1], Thomas Sewell[1], Rob Sison[1], Hira Syeda[3,4], Magnus Myreen[4], Michael Norrish[6], and Gernot Heiser[1]

[1]UNSW Sydney, Australia *({junming.zhao,miki.tanaka,thomas.sewell,r.sison,gernot}@unsw.edu.au)*
[2]ETH Zürich, Switzerland *(alegnani@ethz.ch)*
[3]University of Melbourne, Australia *(hira.syeda@unimelb.edu.au)*
[4]Chalmers University of Technology, Sweden *(myreen@chalmers.se)*
[5]University of Gothenburg, Sweden *(johannes.aman.pohjola@gu.se)*
[6]Australian National University, Australia *(Michael.Norrish@anu.edu.au)*

Abstract

Device driver bugs are the leading cause of OS compromises, and their formal verification is therefore highly desirable. To the best of our knowledge, no realistic and performant driver has been verified for a non-trivial device. We propose Pancake, an imperative language for systems programming that features a well-defined and verification-friendly semantics. Leveraging the verified compiler backend of the CakeML functional language, we develop a compiler for Pancake that guarantees that the binary retains the semantics of the source code. Using automatic translation of Pancake to the Viper SMT front-end, we verify a performant driver for an Ethernet NIC.

1 Introduction

Device driver bugs are the leading cause of OS compromises, accounting for the majority of the 1,057 CVEs reported for Linux in the period 2018–22 [MITRE Corporation, 2023]—clearly they should be the #1 targets of OS verification efforts.

While there have been a number of prior efforts to verify drivers [Alkassar, 2009; Alkassar and Hillebrand, 2008; Chen et al., 2016; Duan, 2013; Duan and Regehr, 2010; Kim et al., 2008; Möre, 2021; Penninckx et al., 2012], to our knowledge none have yet succeeded on realistic, non-trivial devices, nor have they presented any performance analysis of the drivers verified.

Most of these efforts demanded highly manual interactive theorem proving, sometimes requiring drivers to be written and analysed in assembly. On the other hand, attempts to apply more usable methods like model checking [Kim et al., 2008] and automated deductive verification [Penninckx et al., 2012] left significant gaps between the analysed model and the real code.

Moreover, of the above, only Chen et al. [2016] provided an end-to-end verification story that preserved the driver's verified correctness from a driver-appropriate systems programming language down to the binary level. Likewise, with no formal semantics or verified compiler, recent proposals like that of [Chen et al., 2024] to verify drivers written in Rust [Klabnik and Nichols, 2017] have no plan to close the semantic gap to the binary.

Devices are commodities: new ones are created all the time. This means that to be practicable, their verification must have a high degree of usability and automation. But verification of C code is made needlessly expensive by C's complicated semantics. In short, the situation demands a performant systems language with support for usable automated verification, and a means of ensuring the semantics of the verified drivers are preserved down to the binary level.

To this end, we present Pancake, an imperative programming language designed to enable verification of low-level systems code, complete with:

- a *verified compiler* from Pancake to binary that leverages the final stages of the verified CakeML compilation stack [Tan et al., 2019]. Section 2 discusses the overall structure of Pancake's semantics and the compiler, and how the verification of its compiler guarantees that the semantics of drivers and other systems code written in Pancake, once verified, will be preserved down to binary level.

- an *automated deductive verification* front-end for Pancake that leverages the Viper verification framework [Müller et al., 2016a], a middle-end for various SMT solver-based verification back-ends. This takes the form of (1) an annotation syntax for Pancake and (2) a transpiler from annotated Pancake to the Viper intermediate language (IL), which we explain in Section 3.

Using this support, we produce the first verification of a device driver for a non-trivial device, a driver for the

seL4-based LionsOS [Heiser et al., 2025] for a 1 Gb/s Ethernet card that is used in popular Arm SoCs.

Our experience shows that Pancake's automated deductive verification support is usable for those with a systems development background, allowing the verification of critical guarantees for practical drivers (Section 4). A first-year PhD student, with a background of mostly systems development, but not formal verification, was able to complete the verification of our Pancake-based Ethernet driver in about three person-months (after manually transcribing a C implementation into Pancake)—a process that is likely significantly faster when applied to further drivers. This demonstrates Pancake as a viable alternative to C for systems-level code, but one with the advantage of accessible end-to-end verification support.

Our evaluation in Section 5 shows that the verified Pancake driver performs very close to the C version.

We discuss in Section 6 the *trusted computing base* (TCB) and threats to the validity of Pancake's automated deductive verification story and end-to-end semantic preservation guarantees, as well as how we plan to address them with future plans for verification.

2 Pancake Language and Compiler

2.1 The Rationale

While C is the de-facto standard systems language, C's semantics has a number of undesirable properties from a verification standpoint: a complicated memory model, underspecified order of evaluation, and the need to prove the absence of undefined behaviour at almost every step. While the seL4 verification demonstrated that these challenges can be overcome, even when verifying machine code without relying on formal properties of a compiler [Sewell et al., 2013], the cost was high: $350/SLOC just for verifying the C code [Klein et al., 2009], and this cost continues to impact evolution of the kernel. While using a verified compiler [Leroy, 2009] with the verification toolchain VST [Appel, 2011] can help, this has to date not resulted in verified real-world drivers.

Attempts have been made to achieve better systems programming languages by incorporating advanced language features that make certain safety properties hold by construction. For example, Cogent has a linear type discipline that prevents memory leaks [Amani et al., 2016], Rust's borrow checker enforces ownership and lifetimes [Klabnik and Nichols, 2017], and Cyclone incorporates garbage collection and ML-style polymorphism [Jim et al., 2002]. Such advanced features can eliminate whole classes of bugs, or at least reduce bug density, but at the cost of making the language semantics and implementation more complicated. Furthermore, garbage collection introduces unpredictable delays that are highly undesirable in low-level systems code. Yet these approaches still fall short of ensuring full functional correctness, and it is unclear how helpful such advanced language features are in achieving it.

Functional correctness proofs produce (and consume) significantly stronger properties than type systems typically guarantee. A stronger type system could give more useful information, but unless it is so powerful (and so undecidable) as to be a full-featured proof calculus, the information we need will almost always be stronger than what the type system provides.

The information provided by a type system is only useful if the type system is sound, but most practical languages have unverified type systems, or type systems with known soundness bugs. Type systems can be verified [Naraschewski and Nipkow, 1999], but type soundness proofs tend to be delicate, and have subtle interactions with seemingly minor changes to a language. Maintaining a type soundness proof for a living language can significantly bog down development.

Moreover, the safety guarantees of a language only hold if no backdoors are used. But in low-level systems programming it is often necessary to break out of a type-safe environment. For example, device driver code must adhere to hardware-specified data locations, layouts and access protocols. Hence driver code written in safe languages must use significant amounts of unsafe code, effectively escapes to C [Astrauskas et al., 2020; Evans et al., 2020], which mostly eliminates the benefit of using a safe language.

Instead of adding more safety features to a language, which tends to make the semantics more complicated, we believe a simple formal semantics will help lead to simple proofs. Of course, such a formal semantics must exist in the first place: despite years of research [Jung et al., 2018; Kan et al., 2018; Wang et al., 2018; Weiss et al., 2019], there is still no complete formal specification of Rust.

We propose **Pancake** as the solution—a radically minimal language that nonetheless offers a sufficiently expressive interface for writing low-level systems programs, such as device drivers, alongside a number of advantages for formal verification. Most importantly, the language is completely specified by a straightforward formal semantics that fits in a few hundred lines of HOL4 code, with a simple memory model, no notion of undefined behaviour, and no ambiguities in evaluation order.

Pancake is an unmanaged language with no static type system, at a level of abstraction between C and assembly. The data representation and memory model are kept as simple as possible: the only kinds of data are machine words, code pointers, and structs. Programs cannot inspect the stack, which simplifies semantics. All memory is statically allocated; there is no equivalent of `malloc`

and `free`. There are no concurrency primitives—making drivers single-threaded [Ryzhyk et al., 2009a, 2010] significantly simplifies verification, maps well onto the modular design of microkernel-based OSes, and is routinely used for drivers on seL4 without undue impact on performance [Heiser et al., 2022, 2025].

```
fun handle_irq()
{   /@ requires valid_device() @/
    /@ ensures valid_device() @/
    var 1 EIR = get_device_EIR();
    !st32 (REG_BASE + EIR_OFFSET), IRQ_MASK;
    while (true)
    {   /@ invariant valid_device() @/
        var rx_work = EIR & EIR_RXF_MASK;
        var tx_work = EIR & EIR_TXF_MASK;
        if (rx_work) {
            /@ unfold full_heap_access() @/
            rx_return();
            rx_provide();
            /@ fold full_heap_access() @/
        }
        if (tx_work) {
            /@ unfold full_heap_access() @/
            tx_return();
            tx_provide();
            /@ fold full_heap_access() @/
        }
        if (!rx_work) {
            if (!tx_work) { break; }
        }
        EIR = get_device_EIR();
        !st32 (REG_BASE + EIR_OFFSET), IRQ_MASK;
    }
    return 0;
}
```

Listing 1: Pancake code snippet (concrete syntax) with annotations.

2.2 The Language and its Semantics

From a programmer's point of view, Pancake looks and feels like a traditional imperative program language (see Listing 1, ignore the `/@...@/` annotations for now). Figure 1 shows the current abstract syntax of Pancake, divided into expressions (*exp*) and statements (*prog*). The nuts and bolts of a typical Pancake program should be familiar to any programmer. Mutable variables, if statements, while loops and the like – nothing fancy. This is a deliberate design decision: we want Pancake to feel simple and familiar to systems programmers, and minimise the cognitive overhead imposed by exotic language features. In our experience, this has been borne out in practice: systems programmers familiar with C have so far found Pancake easy to learn.

Since Pancake is targeting verified low-level code, another key design concern is to give programmers direct access to low-level details without the language getting

$$
\begin{array}{rcl}
\mathit{exp} & := & \mathsf{Const}\ \mathit{word} \mid \mathsf{Var}\ \mathit{string} \mid \mathsf{Label}\ \mathit{string} \\
& \mid & \mathsf{Struct}\ \mathit{exp}^{\star} \mid \mathsf{Field}\ \mathit{num}\ \mathit{exp} \\
& \mid & \mathsf{Load}\ \mathit{shape}\ \mathit{exp} \mid \mathsf{LoadByte}\ \mathit{exp} \\
& \mid & \mathsf{Op}\ \mathit{binop}\ \mathit{exp}^{\star} \mid \mathsf{Cmp}\ \mathit{cmp}\ \mathit{exp}\ \mathit{exp} \\
& \mid & \mathsf{Shift}\ \mathsf{shift}\ \mathit{exp}\ \mathit{num} \mid \mathsf{BaseAddr} \\
& \mid & \mathsf{BytesInWord} \\
\\
\mathit{prog} & := & \mathsf{Skip} \mid \mathsf{Dec}\ \mathit{string}\ \mathit{exp}\ \mathit{prog} \\
& \mid & \mathsf{Assign}\ \mathit{string}\ \mathit{exp} \mid \mathsf{Store}\ \mathit{exp}\ \mathit{exp} \\
& \mid & \mathsf{StoreByte}\ \mathit{exp}\ \mathit{exp} \mid \mathsf{Seq}\ \mathit{prog}\ \mathit{prog} \\
& \mid & \mathsf{If}\ \mathit{exp}\ \mathit{prog}\ \mathit{prog} \mid \mathsf{While}\ \mathit{exp}\ \mathit{prog} \\
& \mid & \mathsf{Break} \mid \mathsf{Continue} \mid \mathsf{Call}\ \mathit{ret}\ \mathit{exp}\ \mathit{exp}^{\star} \\
& \mid & \mathsf{Raise}\ \mathit{string}\ \mathit{exp} \mid \mathsf{Return}\ \mathit{exp} \mid \mathsf{Tick} \\
& \mid & \mathsf{ShMemStore}\ \mathit{opsize}\ \mathit{exp}\ \mathit{exp} \\
& \mid & \mathsf{ShMemLoad}\ \mathit{opsize}\ \mathit{string}\ \mathit{exp} \\
& \mid & \mathsf{DecCall}\ \mathit{string}\ \mathit{shape}\ \mathit{exp}\ \mathit{exp}^{\star}\ \mathit{prog} \\
& \mid & \mathsf{ExtCall}\ \mathit{string}\ \mathit{exp}\ \mathit{exp}\ \mathit{exp}\ \mathit{exp} \\
& \mid & \mathsf{Annot}\ \mathit{string}\ \mathit{string}
\end{array}
$$

Figure 1: Abstract syntax of Pancake.

in their way. This is part of the motivation for Pancake's perhaps most radical design decision: no static type system, and no distinction between different kinds of data. In sloganeering terms, we might say that Pancake is a language where everything is a machine word. For example, there is no distinction between pointers and integers: it's all words.

We can choose to treat a word as an integer by adding or subtracting with it, and we can choose to treat a word as a pointer by dereferencing it. This means the programmer is free to do arbitrary pointer arithmetic. This is of course unsafe in general, and we do not attempt to make it safe; rather, we give it a simple and well-defined semantics that can support formal verification, without the need for complicated rules about (say) pointer provenance. In this way, the historically minded reader may notice that the language is rather closer to BCPL than C in terms of design philosophy.

With this in mind, the data representation and memory model of Pancake are kept as simple as possible. There are only three kinds of data: *machine words*, *code pointers*, and *structs* (whose fields are machine words, code pointers, or nested structs). Local variables are stack-allocated, and the language does not allow pointers into the stack. Global data may be stored in a statically allocated global memory region: there is no equivalent of `malloc` and `free`.

The operational semantics of Pancake is specified in the same style as CakeML's: *functional big-step semantics* [Owens et al., 2016], which uses an evaluation function from programs to results. In standard relational big-step semantics, a program is given meaning by a *relation* between programs and results. The functional style is similar to a language interpreter, but not necessarily executable. All the intermediate languages used in the compiler have this kind of semantics. This style simplifies

formal proofs of compiler correctness by making the semantics more amenable to term rewriting. The semantics of a program is defined in terms of how it communicates with the outside world; specifically, as a possibly infinite trace of I/O events, each of which denotes either a returning foreign function call or a shared memory load/store operation. The semantics is parameterised on a function that models the effects of these observable I/O events, i.e., how they change the state of the outside world and what data they pass back to Pancake.

2.3 Verified Compiler

The Pancake compiler is verified, and thus eliminated from the TCB. This sidesteps the need for fragile validation of the compiler output on a program-by-program basis [Sewell et al., 2013].

We re-use the lower parts of the verified CakeML compiler [Kumar et al., 2014; Tan et al., 2019]. Figure 2 shows the relationship, with CakeML compiler on the right side and the Pancake compiler on the left. The CakeML compiler consists of 36 (term rewriting) passes, indicated by arrows in the diagram, and uses 11 intermediate languages (coloured boxes). It compiles a strict functional language (in the style of Standard ML and OCaml) all the way down to concrete machine code for six target architectures.

CakeML itself is a high-level functional programming language, unsuited for low-level systems programming in resource-constrained environments where predictable performance is important. CakeML's memory management is all handled by the language runtime, and memory allocation may trigger a stop-the-world garbage collector at any time.

Pancake, in contrast, is explicitly designed to be unmanaged and close to hardware, and, importantly, no runtime. Yet by integration into the CakeML ecosystem, it can reuse many of the existing correctness proofs for the CakeML compiler.

The first few phases of our Pancake compiler goes through two intermediate languages (ILs) that are separate from the CakeML compiler's ILs. The first compiler phase flattens structs and converts the programs to CrepLang, which is a stepping stone into LoopLang. In LoopLang, we compute minimal live sets and divide loops (including their break and continue statements) into tail-recursive functions that better fit the CakeML IL called WordLang. When the program under compilation is translated from LoopLang to WordLang, all loops are replaced with fast tail-calls, as WordLang has no loops. However, the CakeML compiler is set up to compile tail calls into fast simple jumps in the generated machine code.

Once we have entered WordLang, we use the CakeML

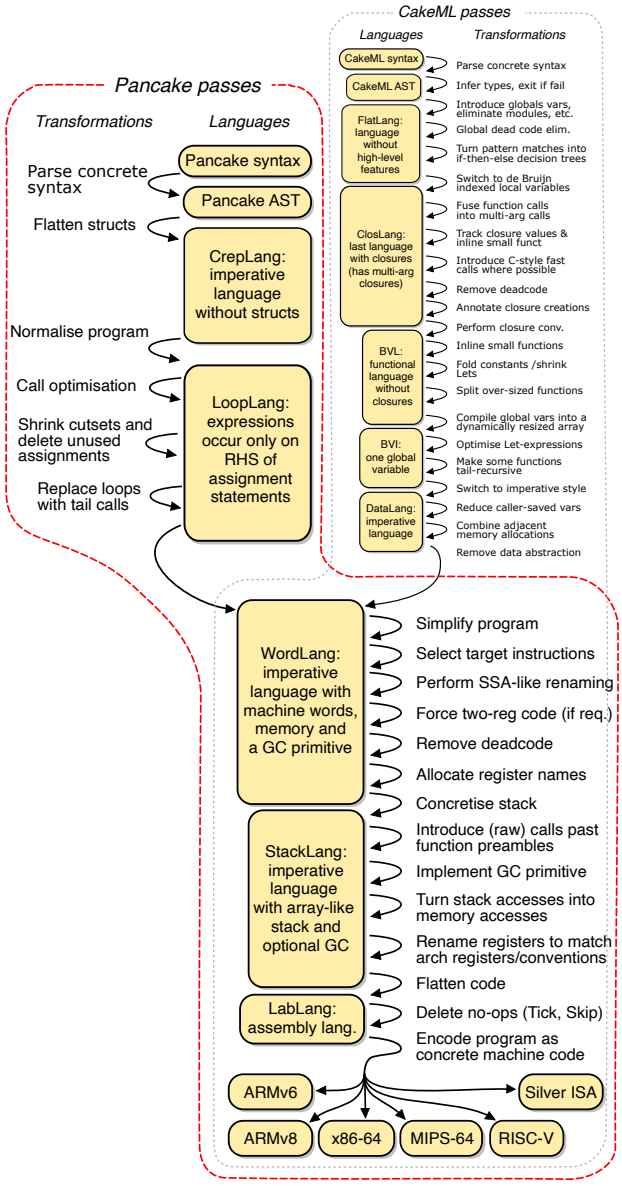

Figure 2: Overview of CakeML and Pancake compiler stack.

compiler's compilation phases, including its phases for instruction selection, register allocation, concretisation of the stack, and, at the end, encoding of the entire program into concrete machine code.

The compiler correctness proofs allow safety and liveness properties of Pancake programs to carry over to the machine code that runs them.

Our Pancake compiler has a special feature by which it not only produces the output machine code, but can also predict the concrete maximum stack size that the program can use, as long as the user's input program did not include any recursive calls that are not in tail position. We have proved that the maximum stack size

that the Pancake compiler returns is sufficient for running the Pancake program without running out of stack space. This allows Pancake to guarantee the absence of premature termination arising from stack overflows.

The functional correctness proof of the Pancake compiler follows the style of the CakeML compiler proofs. The top-level Pancake compiler correctness theorem states that the source Pancake program and the compiler-generated machine code exhibit exactly the same semantics, in particular, the same observable behaviour (I/O events), if the compiler-generated code is run with at least as much stack space as its maximum stack requires.

The compiler and the associated proofs are free software, and available online.[1]

2.4 Challenges with Device Drivers

CakeML assumes exclusive ownership of its statically allocated memory region and that this memory is not observable by the environment. These assumptions are not valid for memory-mapped device registers, which are special memory locations used for interfacing with device hardware, nor does it hold when devices directly write to memory (DMA).

With this limitation, any interaction with the device from CakeML would have to be mediated by FFI calls to C, adding a layer of indirection as part of the TCB. We eliminate this indirection in Pancake by adding native support for interacting with shared memory.

A key challenge here is that the ISA models used by the compiler backend [Tan et al., 2019] are inherently sequential, and 100+ KSLOC of compiler correctness proofs inextricably rely on this fact. Previous work has integrated driver models for specific devices directly into an ARM ISA model with an interleaving semantics [Alkassar et al., 2007]. This approach would have to abandon much of the existing proof base, and would not provide the modularity we need for targeting multiple ISAs and devices. We therefore parameterise the language semantics on a model for shared memory, which supports proof reuse and provides flexibility for incorporating arbitrary devices.

We also add shared memory semantics to all compiler ILs from Pancake downwards, and verify compilation all the way down to machine code, from Pancake abstract syntax which we extend with load/store operations dedicated for shared memory regions (`ShMemLoad` and `ShMemStore` in Figure 1). These operations represent reads and writes that are treated as observable events, just as in the case of any foreign function calls, while being compiled into the load and store instructions of the target machine-language.

Another challenge arising from the CakeML lineage is the issue of function entry points. Implementing the expected driver interface in Pancake requires a mechanism that allows the functions in question to be called from outside the Pancake program. However, the compilation output inherited from CakeML dictates that execution enters through the main function and exits entirely when the main function returns. Additionally, the main function is preceded by lengthy initialisation, and does not support parameters.

Using this execution flow to implement the driver interface demands workarounds, such as branching in the main function based on data indirectly passed through memory, and manually editing the compiler output to return to the caller. Notably, it also results in a substantial performance penalty from repeated re-initialisation. A calling mechanism with this basis is neither scalable nor performant.

Instead, Pancake's support for multiple entry points provides a calling mechanism to address these issues. Using this feature, a function specification can be tagged with the `export` keyword, which extends the generated compiler output to expose the function to the calling conventions of the target platform and restore the state initialised by the main function when the exported function is called. This circumvents the re-initialisation, and handles argument passing and returning to the caller without programmer intervention. It enables calling into the Pancake driver as if it was written in C. The compiler correctness proofs currently account only for the `main` entry point but not for reentry points.

2.5 Verification Approaches

Our aim is high-assurance machine code implementing device drivers. The verified compiler (Section 2.3) guarantees that our source code (written in Pancake) compiles to machine code that implements the source's behaviour. This is half of the story: we must also be sure that the Pancake code we have compiled implements the device driver correctly (incorrect source + verified compiler = incorrect machine code).

The gold-standard (but expensive!) way to verify source code is to use interactive theorem-proving (ITP), as done by (for example) the seL4 [Klein et al., 2014] and CertiKOS [Gu et al., 2016] kernels, and the CompCert [Leroy, 2009], and CakeML [Kumar et al., 2014] (and thus Pancake) compilers. As Pancake is an imperative language, it would be natural to use an ITP-based implementation of Hoare logic as the foundation for the necessary proofs. We describe preliminary and future work on ITP-based approaches to source verification in

[1] `http://code.cakeml.org` for source, or `http://cakeml.org` for pre-packaged versions. Note that Pancake is fully integrated into the CakeML compiler, rather than a stand-alone release.

Section 6.2.2 below.

However, as described in Section 3, our approach here is to use automated deductive verification, rather than an ITP, for productivity reasons. Using Viper [Müller et al., 2016a] (and its realisation of Hoare logic) increases the size of the trusted computing base (see Section 6.1 below), but can be expected to be less expensive in terms of developer time and expertise. The example of our case study bears this out.

Supporting this, Pancake syntax includes Viper annotations (visible in Listing 1). These annotations resemble comments in the source code, and can appear at the top level and within function bodies. Top-level annotations are used, for instance, to specify function contracts, while in-function annotations can be used to specify loop invariants. The Pancake compiler treats annotations as null statements, and they disappear in the first phase of compilation. However, the Viper transpiler (see below) preserves the annotations so that verification can be carried out.

3 A Viper Front-end for Pancake

Our verifier transpiles annotated Pancake code into the Viper IL, then verifies the generated Viper code using Viper's symbolic execution backend Silicon [Schwerhoff, 2016]. This approach is comparable to that of Viper front-ends such as Gobra [Wolf et al., 2021] for Go and Prusti [Astrauskas et al., 2022] for Rust.

Our transpiler first uses the `-explore` option of the Pancake compiler to extract the abstract syntax tree (AST) of the input program, which includes Pancake-level (1) annotations of the kind shown in Listing 1 as well as (2) hooks into a device model that we will describe in Section 3.3. It then translates that AST into encodings we have chosen for Pancake's variables, values, memory locations, and annotated logical assertions over these, as expressed in Viper IL code.

The rest of this section will document and explain the most interesting of these encoding decisions, which regard our encoding of Pancake's machine word type and memory in Viper. Our objective is to choose an encoding that maximises the performance of the resulting queries to Viper's backend, while still being sound—that is, it does not produce a Viper query that is verifiable as true when the assertion from Pancake level that it is supposed to represent is actually false.

We aim for most produced queries to take seconds or minutes to verify, to make verification usable as an active part of a driver developer's workflow. Our overriding concern is to prevent the transpiler from producing queries that cause the Viper SMT-based back-end either to diverge or to take a prohibitively long time to be used for continuous integration testing. We leave proving the soundness of the encoding to future transpiler verification work (see Section 6).

3.1 Machine-Word Encoding

For the best query performance, we encode Pancake's machine-size word variables as integers in Viper.

As a language whose only primitive type is machine-size words (used for both values and pointers), Pancake's word variables have bitvector semantics and overflows are well-defined behaviour, i.e., the variable is wrapped modulo the word size.

To preserve these semantics when encoding these as integers in Viper, which are signed and unbounded, one option is to treat arithmetic operations as modulo the word size. However, this approach significantly slows verification due to the mixing of arithmetic that is linear (such as addition, subtraction etc.) versus non-linear (such as modulo and bitvector operations) with respect to Viper's unbounded signed integer space. Poor query performance due to mixing of different theories that require handling by different solver strategies is a well-known hazard for SMT solvers [Jovanovic and Barrett, 2013].

Instead, since overflows are rarely intended behaviour, we adopt Prusti's approach, treating overflows as verification failures. We do this by checking the bounds of every variable after having unrolled all arithmetic operations into three-address code. We make this decision to ensure soundness at the mild expense of disallowing intended machine word overflows. These were used in the original C implementation of the driver but could be avoided with a one line change in the Pancake version.

Another concern is that bitvector operations are also non-linear with respect to Viper integers, causing similar performance concerns. Observing that almost all bitvector operations in our driver occur as part of bit masking, shifting etc. for accesses to device memory as part of a well defined device interface, we instead abstract this device interface, which we cover in Section 3.3. Additionally we precompute constant expressions and apply heuristics to rewrite common bitvector operations, e.g. `x&255` is rewritten as `x%256`. Although the rewritten operations still use non-linear arithmetic, performance is improved compared to bitvector operations. This eliminates the vast majority of bitvector operations, with the remaining few in our Ethernet driver turning out to be reasonably performant.

3.2 Local Memory Encoding

As mentioned in Section 2, Pancake disallows pointers to stack variables, which simplifies modelling them in Viper, as for these we do not have have to manage any

kind of access permission or to check for references to invalid memory.

For all other memory, the memory model of Pancake poses unique challenges due to its assembly-like nature, such as the lack of first-order support for arrays and reliance on pointer arithmetic for memory operations. Due to the lack of information about what different memory regions represent—be it an array, struct or other data structures—we must adopt the naivest modelling approach: with the exception of shared and device memory (described in the next section), we encode memory as an array of words. This approach results in non-idiomatic Viper code but captures Pancake's native word-size treatment memory operations accurately.

The various memory regions a function needs access to, and with which permission, are added via annotations. These annotations are encoded as an iterated separating conjunction [Müller et al., 2016b], which can be verified efficiently in Viper.

3.3 Shared and Device Memory Encoding

As mentioned in Section 2.4, Pancake includes dedicated load/store operations for shared memory regions, whose behaviour in a sense resembles the effect of the `volatile` keyword in C which prevents the compiler from reordering or optimizing away those accesses. Thus, our driver uses these primitives for accesses to device memory, as well as accesses to memory shared between it and other OS components. Moreover, unlike the rest of memory, we cannot encode shared memory in Viper simply as an array of words, because we cannot rely on its contents not to change between accesses.

Device register accesses, needed for implementing drivers, make extensive use of bitwise operations to access the correct bits. As stated before, these operations result in bad performance so we seek to avoid them where possible.

```
1  method store_rx_free(heap: IArray,
2      device: Ref, addr: Int, value: Int)
3
4  /@ shared rw u64 rx_free[lower..upper] @/
```

Listing 2: Method signature of shared memory store in device model and the corresponding top-level annotation in Pancake, specifying that shared memory accesses to the address range `lower..upper` should transpile to a Viper invocation of `store_rx_free` or `load_rx_free`.

To limit these, we model accesses to shared memory as separate Viper method calls, which the driver developer should specify in an external Viper file representing the *device model* for the driver's target device. These methods define valid operations for specific address ranges corresponding to particular device registers and memory regions. They also specify `requires` and `ensures` clauses for the hardware interfaces, as assertions in terms of a global device state and the non-device memory. This is a good fit with how the Pancake semantics models shared memory operations (Section 2) as observable events, whose interpretation is parameterised on a model of the environment—a semantics which is also preserved by the Pancake compiler.

For interactions via shared memory with other OS components, we make it the responsibility of the driver developer to specify a *neighbouring component model* similarly in a separate Viper file. The developer should use this to capture the guarantees the driver should meet for the Viper verification to enforce, as well as any assumptions about the behaviour of those neighbouring components with respect to the shared memory.

We then provide a syntax for top-level Pancake annotations that allow the driver developer to specify the correct method(s) to use for a shared memory operation, according to the address range the driver interacts with, allowing the transpiler to infer automatically which Viper method the produced Viper model should invoke in place of the shared memory interaction (see Listing 2).

This way, the driver developer does not necessarily have to specify the model of the device or neighbouring component before they implement their driver in Pancake, and furthermore they do not have to modify their Pancake driver's shared memory accesses after specifying these models—they just add extra annotations to specify which Viper methods the transpiler should produce in place of loads/stores to given shared memory addresses. This approach ensures the separation of driver implementation and device specification, whilst also improving verification speeds.

4 Verified Ethernet Driver

We formally verify a single-core i.MX Ethernet driver for LionsOS [Heiser et al., 2025]. LionsOS uses a simple OS-side interface for device drivers using zero-copy shared-memory communication for network data, and lock-free, bounded, single-producer, single-consumer (SPSC) queues for meta data and control information. The driver synchronises with the rest of the OS via semaphores (implemented as seL4 Notifications).

The target driver, implemented in Pancake, controls the MAC-NET 1 Gb/s Ethernet core common to NXP i.MX 8M Mini, Dual, QuadLite and Quad Applications Processors present in various Arm-based NXP system-on-chips (SoC), operating as a network interface card (NIC). The NIC uses DMA descriptor rings for passing the addresses of data buffers.

The driver incorporates formal specifications through annotations using Hoare logic and the Viper verification framework. This verification work establishes four classes of critical guarantees over the device driver, which we will examine in turn: (1) protocol compliance with NIC device interfaces (Section 4.1), (2) protocol compliance with neighbouring OS component interfaces (Section 4.2), (3) guaranteed data integrity across transfers (Section 4.3), and (4) memory safety through region separation and restricted access controls (Section 4.4).

4.1 Device Model and Properties

We model and enforce three kinds of properties with respect to the driver's correct use of the device interfaces: (1) that device memory accesses are within the right address ranges, (2) that values written to the device obey the device's requirements, and (3) that verification of the rest of the driver is robust for any values read from the device, as constrained only by the device model.

First, we enforce that the driver only accesses the parts of device memory that comprise the NIC's hardware interfaces for packet receipt and transmission— namely, the device's RX (receive) and TX (transmit) hardware descriptor ring regions and other essential registers. To enforce this, recall from Section 3.3 that our transpiler supports and looks for top-level Pancake annotations that specify which addresses correspond to valid device interfaces, as illustrated in Listing 2. Our verifier will reject any Pancake drivers that attempt to invoke a shared memory load or store operation on an address that is not covered by any such annotations.

```
1  method store_EIR(device: Ref, addr: Int,
2      value: Int)
3    requires addr == (REG_BASE + EIR_OFFSET)
4    requires value == IRQ_MASK
5    requires valid_device(device)
6    ensures valid_device(device)
7
8  method load_EIR(device: Ref, addr: Int)
9    returns (retval: Int)
10   requires addr == (REG_BASE + EIR_OFFSET)
11   requires valid_device(device)
12   ensures bounded32(retval)
13   ensures valid_device(device)
14
15 /@ shared rw u32 EIR[REG_BASE + EIR_OFFSET] @/
```

Listing 3: Examples of device register store/load interfaces in the device model as specified by Viper methods using `requires` and `ensures`, and corresponding top-level Pancake annotation.

Second, we verify that whenever the driver interacts with these device interfaces, it does so in the required way not to put the device into a bad state, as specified by its documentation and captured by our device model. To enforce this, we specify two kinds of `requires` clauses for device interface methods, as illustrated in Listing 3 for a representative pair of examples, the `store` and `load` methods for `EIR`, a particular device register:

1. Method-specific requirements, such as the `store_EIR` method's `requires` of both an *address* and a *value* requirement—namely, that the address is the `EIR`'s and that the driver only ever writes a particular `IRQ_MASK` constant to it;
2. Device-wide invariants—in this example, captured by `valid_device` in `store_EIR`'s third `requires` clause. For our NIC, `valid_device` asserts that the state of the hardware descriptors remains valid: the bitfields are cleared and set properly according to the device's documented specifications, data pointers are 32-bit width and byte-aligned, and data lengths are within 16-bit bounds.

The device state of our NIC device, as modelled by `device` and asserted valid by `valid_device(device)`, comprises the hardware descriptor rings, with each ring decomposed into three integer sequences representing data addresses, lengths, and bitfields. These integers represent machine words and are converted by Viper to bitvectors when bit-level operations are required. Our decomposition of these descriptor components into native Viper integers thus simplifies verification and reduces SMT solver complexity, particularly when using non-linear arithmetic and bitvector operations.

Note that, while the presence of `valid_device` in the `requires` clauses of device interface methods requires the driver not to violate the validity of the device state, its presence also in the `ensures` clauses of all device methods specifies that we can assume the device itself will maintain that same validity invariant throughout all driver-device interactions.

Finally, we model the non-determinism of the values the driver could possibly obtain from the device by underspecifying its interface methods, forcing the verification of the calling context of the method (i.e. the Pancake code that invoked the shared memory operation) to account for a wide range of possible values constrained only by their `ensures` clauses. For example, `load_EIR` in Listing 3 ensures that the return value is a valid unsigned 32-bit word value, as captured by `bounded32(retval)`. Verification must subsequently succeed for any value returned by that interface that satisfies the `ensures`, i.e. any unsigned 32-bit word value.

4.2 OS Communication Protocols

Using much the same techniques just described in the previous section for specifying valid device interactions,

we constrain the driver's access to shared memory regions for network SPSC queues through annotations when interfacing with LionsOS components. We also model the shared SPSC queues non-deterministically to verify that the driver maintains protocol compliance—assuming the neighbouring OS component maintains it too—without assuming specific state values.

We verify that the driver adheres to the following network queue signaling protocols: When consuming data buffers from network queues, the driver requests wake-up signals from the OS when hardware rings have vacancy. When providing data buffers to the OS, the driver signals the OS's semaphores if and only if (1) a signal was explicitly requested by the OS and (2) the queue state has changed by the driver [Heiser et al., 2025]. To prevent double signaling, we also ensure that the driver clears signal requests after notification. We also verify that queue states remain valid during all SPSC queue operations, for instance we check that no queue overflows or underflows occur at the driver side.

4.3 Data Integrity

To ensure reliable data transfer between the OS and the device during the translation between hardware descriptor and SPSC formats, we verify that the driver maintains data integrity by tracking packet addresses and lengths.

For example, we check that the given data address and data length are stored properly after updating the TX hardware descriptor ring, as shown in Listing 4.

```
buffer = net_dequeue(os_tx_avail);
update_tx_hw_ring(hw_tail, buffer);
/@ assert(device.hw_ring_tx[hw_tail].data_addr
      == buffer.data_addr) @/
/@ assert(device.hw_ring_tx[hw_tail].data_len
      == buffer.data_len) @/
```

Listing 4: Data integrity verification example in annotated Pancake. For brevity, unwrapping of predicates referred to by the assertions is omitted.

We establish this integrity check on all pathways of data transfers in the driver. We also verify data transfer completeness by ensuring that within the driver, the number of SPSC queue operations align with the number of hardware descriptor ring state changes, so that there is no data loss in the driver.

4.4 Memory Access Control

In addition to the memory access constraints described in Section 4.1 and Section 4.2, we furthermore verify that only the parts of the driver responsible for packet transmission paths access any TX-related descriptor rings and

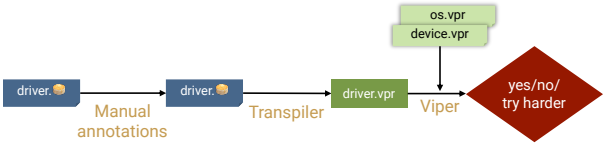

Figure 3: Driver verification workflow.

SPSC queue state, and likewise that only its packet receipt paths access RX-related state.

We enforce this using Viper's native permissions features to specify access controls, in this way providing formal verification of memory safety and region isolation. In effect, our driver verification applies a separation logic-like principle by partitioning the driver's global memory into RX and TX regions (which reflects their use in LionsOS [Heiser et al., 2025]).

4.5 Verification Workflow

Figure 3 shows the resulting verification workflow. The verifier annotated the Pancake source of the driver and processes the resulting source with the transpiler, which produces the input for Viper. The verifier also supplies Viper specifications of the device interface, as well as the interface between the driver and the rest of LionsOS. These are then processed by Viper which either returns a result (proved or falsified) or times out.

5 Performance Evaluation

We now examine how our verified Pancake driver compares to the original C implementation.

Pancake driver compilation time is a matter of seconds, and the verification of the driver in full takes around 20 minutes on a typical laptop. When verified separately, the device model takes 10 minutes and the driver's functions each take around 1 minute.

Our evaluation platform is an AVnet MaaXBoard with an NXP i.MX8MQ SoC, having four Arm Cortex A53 cores capable of a maximum of 1.5 GHz; we run our measurements at a fixed clock rate of 1 GHz. The board has 2 GiB of RAM and the on-chip 1 Gb/s NIC specified earlier (in Section 4).

The evaluation system runs a networking client on LionsOS. The client simply receives data packets from the NIC and echoes them back. We use an external load generator that sends an adjustable load (requested throughput) to the target system, and measures the amount of data received back (received throughput) as well as the latency. On the evaluation system we also measure CPU load.

Figure 4 shows the result. The system has no problems handling the requested load: the received through-

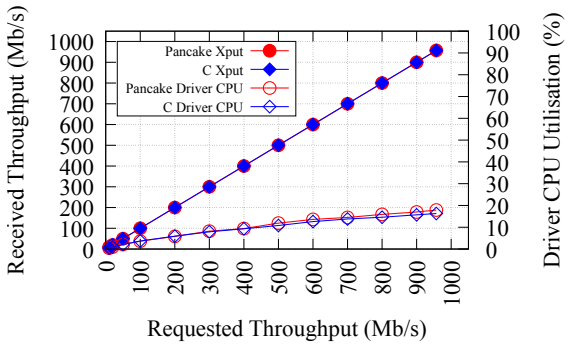

Figure 4: Performance of Ethernet Driver written in Pancake vs C, in terms of achieved throughput (Xput) and Driver CPU utilisation.

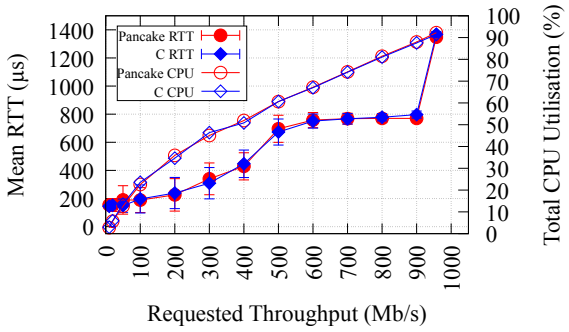

Figure 5: Performance of Ethernet Driver written in Pancake vs C, in terms of average round trip time (RTT) with standard deviation and total CPU utilisation.

put equals the requested. The Pancake version of the driver uses slightly more CPU time to handle the load than the C version, the difference is about 10%.

Figure 5 puts this slight increase into context, by looking at latency (RTT) and the *overall* CPU use of the whole system. The difference in CPU usage of the drivers becomes unnoticeable. Similarly, the differences in latency is well within the variance of the results. We can summarise that the performance cost of the verified driver is in the noise.

6 Discussion

While our main results are formally verified, no proof about a real-world artifact is ever fully complete and self-contained: there will always be a *trusted computing base* (TCB). The TCB, in brief, is everything that we rely on for the correct operation of our system, but which is currently outside the scope of the formal verification effort. For a formal verification result, the TCB is the main threat to validity.

In Section 6.1, we discuss our TCB. Section 6.2 discusses alternative verification approaches we are pursuing that would mitigate the TCB. Finally, Section 6.3 discusses potential usability improvements to the verification front-end.

6.1 Threats to Validity

For the Pancake compiler correctness proofs, we trust: that the HOL4 theorem prover is a sound implementation of higher order logic; that the official specification of the Arm ISA is correctly implemented by the CPU [Kanabar et al., 2022]; and that an unverified linker for connecting the Pancake binary to external code such as LionsOS library routines is correct. With these as the TCB, the Pancake compiler is verified, just like the CakeML compilation passes that it leverages, to preserve the semantics of the `main` function of the source program.

Our verification results presented here establish certain observable properties of the driver including its behaviour after reentry. Putting these together, the verified compiler guarantees that the binary obtained by compiling this driver should preserve the verified observable properties, although we are not (yet) providing links between the two results formally, in the sense that there is no theorem yet to state this on a single, unified formalism. Future work will strengthen this by verifying the transpiler as well as extending the compiler proofs to account for reentry points.

Moreover, while we plan to verify our transpiler from annotated Pancake to Viper (Section 6.2.1), this is not done yet. Until then, a mistake in the transpiler could produce a Viper query that the underlying SMT-based backend can prove true, even if the property as specified at the Pancake program point via Viper annotations is false. Beyond this, we also trust the Viper verification infrastructure and SMT solvers called by its backends to discharge the Hoare logic queries specified using its input language soundly, i.e. only when true.

Like any device driver, we trust the device not to malfunction—that is, to guarantee that it meets the `ensures` specifications we impose on return from device interfaces in our device model. This includes trusting that the device's initialisation process establishes a valid initial state. Verification of the device hardware and initialisation process would be needed to gain further assurance of this.

Meanwhile, we assume that the neighbouring OS components comply with the SPSC queue protocols, as mentioned in Section 4.2, and that the driver acts as the sole consumer or producer per queue. We will need to leverage Viper's access permission system to establish more sophisticated guarantees about concurrent accesses and thread safety properties.

Finally, we trust the operating system kernel and its

userland support libraries not to crash or malfunction. LionsOS runs on the seL4 OS microkernel, which is verified not to crash [Klein et al., 2014], and relies on the seL4 Microkit support library, whose main server loop has been verified not to exhibit undefined behaviour [Paturel et al., 2023]. However, functional properties of individual seL4 system calls relied on by Microkit have not yet been verified.

6.2 Future Verification Efforts

6.2.1 Transpiler

Work towards a formal correctness proof for the implementation of the transpiler described in Section 3 is still ongoing. Though the Viper IL code is obtained by the transpiler in a straightforward manner from the original Pancake code, presently we have no formal proof of a semantic correspondence between the input Pancake code and the resulting Viper IL code. Consequently, the transpiler is currently included as part of the trusted computing base, as discussed in Section 6.1.

Such a proof would complete our end-to-end verification story, by allowing us to soundly infer the correctness of our initial Pancake code from a successful verification run of the corresponding Viper IL code. This assumes the correct operation of the Viper toolchain which our verification step relies on. Existing efforts to validate parts of this toolchain can be found in [Gössi, 2016; Parthasarathy et al., 2024].

Our eventual plan is to complete a mechanised correctness proof for the Pancake to Viper IL transpiler within the HOL4 theorem prover. As a first step towards this, we require formal descriptions of the transpiler, and the semantics of Pancake and the Viper IL. Though the current Rust implementation of the transpiler contains 7 kLOC, a significant portion of this code consists of workarounds which extract the underlying abstract syntax from an unparsed Pancake program without interfacing directly with the existing Pancake parser. However, as part of the CakeML project, the Pancake language already has its parser defined in HOL4. Additionally, the formal syntax and semantics of Pancake have been mechanised in HOL4. Thus, an implementation in HOL4 can access the abstract syntax directly, which could simplify the implementation considerably.

6.2.2 Interactive Theorem Proving for Pancake programs

The Pancake language semantics are defined in the HOL4 interactive theorem prover (ITP). It would be natural to verify a Pancake program directly in HOL4, proving that its semantics agree with its specification: it would significantly shrink the TCB. Furthermore, it would be possible to formally compose the source-level verification with the compiler correctness proof for Pancake (also in HOL4) to give an end-to-end proof that the compiled binary is correct.

Hoare logic We have defined a Hoare logic for Pancake programs in HOL4. This is a standard approach to verifying programs in an interactive language, and resembles the proof approach of Viper. Programs are decorated with preconditions, postconditions, and loop invariants. The logic specifies how a proof of a precondition/-postcondition pair for a function body can be decomposed into proofs about each statement. We have defined the logic in HOL4 and proved its soundness against the Pancake semantics. We have also added an automated proof tactic that drives the logic, doing all the decomposition and re-composition steps, leaving the user to prove verification conditions such as that the precondition they supplied is sufficient, that their loop invariant implies itself at the next iteration, etc.

At the time of writing, this Hoare logic automation makes the proof process far more productive than a naive manual approach. But our proof productivity with this automation is still far short of what is achieved with our Viper-based approach. We think that there is still substantial room for improvement in future work.

One interesting possibility is to try to maximise compatibility between the HOL4 Hoare logic and the fragment of Viper's annotation syntax we are using in this work. The aim would be that an initial verification in Viper could later be minimally converted into a HOL4 verification, and replayed without the involvement of Viper or the transpiler. This is a potential alternative to verifying the transpiler and would also take Viper and the SMT solvers out of the TCB.

Interaction trees semantics As mentioned in Section 2.2, Pancake programs that communicate with the outside world are parameterised on a function that models how the outside world behaves. These models are deterministic, which simplifies definitions and compiler proofs but makes modelling realistic devices awkward: from a programmer's point of view, devices are non-deterministic. Moreover, requiring device models upfront makes it difficult to decouple reasoning about code and about devices.

A more promising approach is interaction trees [Xia et al., 2020], which represent the behaviour of a program as a coinductive tree, with nodes for actions and branches for possible environmental responses. This model naturally accommodates non-determinism, in contrast to the linear-time view of a functional semantics.

We have developed an interaction tree semantics for Pancake and are currently proving its correspondence to

the functional semantics. We are also currently applying it to verification of device drivers similar to those described here.

6.3 Transpiler usability improvements

The current workflow involves transpiling Pancake into Viper for verification. Whilst the transpiler allows for direct verification, error messages are tied to the generated Viper code and are not reflected back to the Pancake code, complicating debugging. Additionally, while many annotations are automatically inferred, some require manual specification, increasing the effort required from the programmer.

In addition to the command line tool we have built an initial development environment for verified Pancake based on Microsoft's VS Code framework. As of yet this lacks some features like support for the separate Viper device model. The transpiler supports verifying functions individually. This could be integrated into the development environment to allow for efficient re-verification of only modified functions.

Despite these shortcomings, we expect that these tools can be combined and improved to form a cohesive and usable toolchain for writing and verifying device drivers.

7 Related Work

A number of prior works have investigated verification of device drivers at various programming language levels. Within this space, researchers have employed different verification approaches like model checking [Kim et al., 2008] and interactive theorem proving [Möre, 2021]. Among these efforts, [Penninckx et al., 2012] developed their verification using VeriFast [Jacobs et al., 2010], a deductive verification approach similar to ours. They also notably extended their analysis to include concurrency properties beyond our current scope. However, a common limitation across all these approaches was the significant gap between the analysed model and the actual executable code—a gap which we narrow by using Pancake's verified compilation stack.

Others, which we detail below, have done better at closing such gaps (e.g. in some cases providing support for direct access by drivers to memory-mapped device ports, as we have), but failed to demonstrate scalability beyond the simplest serial drivers – among other reasons, by failing to include any performance evaluation of the drivers they verified.

The earliest driver verification effort for a non-trivial device we are aware of is that of Alkassar [2009]; Alkassar and Hillebrand [2008], who verified a (still simplified) ATAPI hard disk device driver in Isabelle/HOL interactive proof assistant [Nipkow et al., 2002] as part of the Verisoft project. Similar to our work, they verify their driver relative to a functional model of the memory-mapped device – in their case, based on a subset of the ATAPI standard. However, the type safety of the fragment of C they used for most of their OS limited their ability to model direct access to memory-mapped device ports directly from that language; consequently, they instead had to write and verify their driver in a MIPS-like assembly language.

Duan and Regehr [2010] presented a framework for verifying device drivers integrated with the L3 model of ARM machine code [Fox, 2003; Fox and Myreen, 2010] for HOL4 [Slind and Norrish, 2008], with a UART driver as the case study. Like Alkassar and Hillebrand [2008] they did not support reasoning about DMA, but Duan [2013] later added support in the form of Hoare triples for device memory access scenarios. The way we integrate shared memory access in Pancake with the specification of `requires` and `ensures` as Viper annotations is similar, and allows us to impose the requirements of our device model on our driver's device memory access directly from the Pancake language. Schwarz and Dam [2014] further extend the L3 model to support device drivers with DMA. This goes further than the device model of our paper, as the Ethernet device we verify is documented not to interfere with the hardware ring indices, which are left under the control of the driver.

As part of the CertiKOS project, Chen et al. [2016] added support for verifying drivers and integrating them with their OS verification framework in the Rocq (formerly Coq) interactive proof assistant. Unlike the above works, whose drivers were implemented in assembly, the serial and interrupt controller device drivers Chen et al. [2016] verified are implemented in ClightX [Gu et al., 2015], an extension of the CompCert Clight language [Leroy, 2009] with extra instrumentation to support CertiKOS's abstraction-guided approach to OS verification. Like our work, their use of a verified compiler (a modification of CompCert) to compile the driver down to binary gives some assurance that any properties proved at the driver source level are preserved down to the binary. However, driver verification in their framework requires interactive proofs in Rocq, for a C variant whose proof relies on adding abstract state elements that can influence program execution. This is more disruptive to the original code than mere annotations or typical "ghost state", and arguably requires more formal methods experience than automated deductive verification via annotations.

Unlike our work, none of the works above presented any analysis or discussion their drivers' performance.

We are also aware of some current efforts by Chen et al. [2024] to verify device drivers written in Rust [Klabnik and Nichols, 2017] using the Verus automated deductive verifier [Lattuada et al., 2023]. However, with-

out a formal semantics let alone a verified compiler, the possibility of end-to-end assurances for Rust-based drivers still seems remote.

Finally, there has been also been work on driver synthesis by [Ryzhyk et al., 2009b, 2014] that took as input detailed specifications of interfaces for (1) the device class the driver needs to implement, the (2) device itself and (3) OS service it needs to provide to the rest of the OS, written in a custom specification language. Although, like in our work, their device interface included details such as valid registers and their sizes, it also included more detailed elements like a state transition diagram. In our work, we have a model of device state that we use for specifying and verifying the maintenance of invariants (the `valid_device(device)` assertion seen in Listing 3 and explained in Section 4.1))—this could in future form the basis for more detailed, state machine-based specifications of internal device states. Note, however, that this synthesis work could not deal with DMA.

8 Conclusion

This paper presents, to our knowledge, the first formal verification of a demonstrably performant driver for a realistic, non-trivial device, the Ethernet NIC common to a number of variants of NXP i.MX 8M processors.

It also introduces the Pancake systems programming language, designed especially for systems-level code to be amenable to formal verification. With Pancake, it makes two enabling contributions: (1) a verified compiler that carries the semantics of Pancake down to binary, leveraging CakeML's verified compiler backend; and (2) an automated deductive verification front-end that takes Pancake with Viper annotations, leveraging the Viper SMT-based verification framework.

This work shows that Pancake is usable for developing verified, performant drivers. A PhD student with a systems background and not much formal methods experience was able to write and verify the aforementioned Ethernet driver in a few person-months. The Pancake driver shows performance very close to C.

This work paves the way for verified development of performant device drivers—a leading source of OS vulnerabilities—as common-place infrastructure.

Acknowledgements

Many thanks to Alessandro Legnani's project co-supervisors Toby Murray, for his role in envisioning the use of Viper, and Peter Müller for his feedback on this. We thank also Krishnan Winter, Benjamin Nott, Craig McLaughlin and Remy Sasseau for their past and ongoing contributions to other aspects of Pancake. Finally, we thank Isitha Subasinghe and Zoltan Kocsis, for their help understanding SMT performance issues from theory mixing.

The development of Pancake was made possible through the generous support of multiple organisations: the UAE Technology Innovation Institute (TII) under the *Secure, High-Performance Device Virtualisation for seL4* project, DARPA under prime contract FA 8750-24-9-1000, and UK's National Cyber Security Centre (NCSC) under project NSC-1686. Finally, Junming Zhao gratefully acknowledges the topup scholarship donated by the winners of the 2023 ACM Software System Award.

References

E. Alkassar, M. Hillebrand, S. Knapp, R. Rusev, and S. Tverdyshev. Formal device and programming model for a serial interface. In *International Verification Workshop*, pages 4–20, Bremen, DE, July 2007.

Eyad Alkassar. *OS Verification Extended – On the Formal Verification of Device Drivers and the Correctness of Client/Server Software*. PhD thesis, Saarland University, Computer Science Department, 2009.

Eyad Alkassar and Mark A. Hillebrand. Formal functional verification of device drivers. In *Verified Software: Theories, Tools and Experiments*, volume 5295 of *Lecture Notes in Computer Science*, pages 225–239, Toronto, Canada, October 2008. Springer.

Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–188, Atlanta, GA, USA, April 2016.

Andrew W. Appel. Verified software toolchain. In *European Symposium on Programming*, volume 6602 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2011.

Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How do programmers use unsafe Rust? *Proceedings of the ACM on Programming Languages*, 4(OOPSLA): 136:1–136:27, 2020.

Vytautas Astrauskas, Aurel Bílỳ, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J Summers. The Prusti project: Formal verification for Rust. In *NASA Formal Methods Symposium*, pages 88–108. Springer, 2022.

Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible OS kernels and device drivers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 431–447, 2016.

Xiangdong Chen, Zhaofeng Li, Jerry Zhang, and Anton Burtsev. Veld: Verified Linux drivers. In *Workshop on Kernel Isolation, Safety and Verification*, page 23–30, New York, NY, USA, 2024. ACM.

Jianjun Duan. *Formal Verification of Device Drivers in Embedded Systems*. PhD thesis, University of Utah, USA, 2013.

Jianjun Duan and John Regehr. Correctness proofs for device drivers in embedded systems. In *Systems Software Verification*, Vancouver, BC, CA, October 2010. USENIX Association.

Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. Is Rust used safely by software developers? In *International Conference on Software Engineering*, pages 246–257, 2020.

Anthony Fox. Formal specification and verification of ARM6. In *International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 25–40, Rome, Italy, September 2003. Springer.

Anthony Fox and Magnus Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *International Conference on Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 243–258, Edinburgh, UK, July 2010. Springer.

Cyrill Martin Gössi. A formal semantics for Viper. Master's thesis, Master thesis, ETH Zürich, 2016.

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 595–608. ACM, 2015.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 653–669, Savannah, GA, US, November 2016. USENIX Association.

Gernot Heiser, Lucy Parker, Peter Chubb, Ivan Velickovic, and Ben Leslie. Can we put the "S" into IoT? In *IEEE World Forum on Internet of Things*, Yokohama, JP, November 2022.

Gernot Heiser, Ivan Velickovic, Peter Chubb, Alwin Joshy, Anuraag Ganesh, Bill Nguyen, Cheng Li, Courtney Darville, Guangtao Zhu, James Archer, Jingyao Zhou, Krishnan Winter, Lucy Parker, Szymon Duchniewicz, and Tianyi Bai. Fast, secure, adaptable: LionsOS design, implementation and performance, January 2025. URL `https://arxiv.org/abs/2501.06234`.

Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 6461 of *Lecture Notes in Computer Science*, pages 304–311. Springer, 2010.

Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX*, pages 275–288, Monterey, CA, USA, June 2002. USENIX.

Dejan Jovanovic and Clark W. Barrett. Being careful about theory combination. 42(1):67–90, 2013.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 66:1–66:34, December 2018.

Shuanglong Kan, David Sanán, Shang-Wei Lin, and Yang Liu. K-Rust: An executable formal semantics for Rust. *CoRR*, abs/1804.07608, 2018. URL `http://arxiv.org/abs/1804.07608`. Preprint.

Hrutvik Kanabar, Anthony C. J. Fox, and Magnus O. Myreen. Taming an authoritative Armv8 ISA specification: L3 validation and CakeML compiler verification. In *International Conference on Interactive Theorem Proving*, page 20:1–20:22, August 2022.

Moonzoo Kim, Yunja Choi, Yunho Kim, and Hotae Kim. Formal verification of a flash memory device driver – an experience report. In *SPIN Workshop on Model Checking Software*, volume 5156 of *Lecture Notes in Computer Science*, pages 144–159, Los Angeles, CA, US, 2008.

Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2017.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.

Ramana Kumar, Magnus Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–191, San Diego, January 2014. ACM.

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1), April 2023. URL `https://doi.org/10.1145/3586037`.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

MITRE Corporation. Linux $\gg$ linux kernel: Security vulnerabilities (CVSS score $\geq$ 9), 2023. URL `https://www.cvedetails.com/vulnerability-list.php?vendor_id=33&product_id=47&version_id=&page=1&hasexp=0&opdos=0&opec=0&opov=0&opcsrf=0\&opgpriv=0&opsqli=0&opxss=0&opdirt=0&opmemc=0&ophttprs=0&opbyp=0&opfileinc=0&opginf=0&cvssscoremin=9&cvssscoremax=0&year=0&month=0&cweid=0\&order=1&trc=3034&sha=544260ec3a86a7e17f8b02b39d6342815d8d4bd5`. Accessed: 2023-01-25.

Tomas Möre. *Formal verification of device driver monitors in HOL 4*. Masters thesis, School of EECS, KTH, SE, 2021.

Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *International Conference on Verification, Model Checking and Abstract Interpretation*, pages 41–62, St. Petersburg, FL, US, January 2016a. Springer.

Peter Müller, Malte Schwerhoff, and Alexander J Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In *International Conference on Computer Aided Verification*, pages 405–425. Springer, 2016b.

Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning*, 23(3-4):299–318, 1999. URL `https://doi.org/10.1023/A:1006277616879`.

Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

Scott Owens, Magnus Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In *European Symposium on Programming*, page 27, Eindhoven, The Netherlands, April 2016.

Gaurav Parthasarathy, Thibault Dardinier, Benjamin Bonneau, Peter Müller, and Alexander J Summers. Towards trustworthy automated program verifiers: Formally validating translations into an intermediate verification language. *Proceedings of the ACM on Programming Languages*, 8(PLDI):1510–1534, 2024.

Mathieu Paturel, Isitha Subasinghe, and Gernot Heiser. First steps in verifying the seL4 Core Platform. In *Asia-Pacific Workshop on Systems (APSys)*, Seoul, KR, August 2023. ACM.

Willem Penninckx, Jan Tobias Mühlberg, Jan Smans, Bart Jacobs, and Frank Piessens. Sound formal verification of Linux's USB BP keyboard driver. In *NASA Formal Methods Symposium*, volume 7226 of *Lecture Notes in Computer Science*, 2012.

Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *EuroSys Conference*, pages 275–288, Nuremberg, DE, April 2009a.

Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with Termite. In *ACM Symposium on Operating Systems Principles*, pages 73–86, Big Sky, MT, US, October 2009b.

Leonid Ryzhyk, Yanjin Zhu, and Gernot Heiser. The case for active device drivers. In *Asia-Pacific Workshop on Systems (APSys)*, pages 25–30, New Delhi, India, August 2010.

Leonid Ryzhyk, Adam Christopher Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. User-guided device driver synthesis. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 661–676, Broomfield, CO, USA, October 2014.

Oliver Schwarz and Mads Dam. Formal verification of secure user mode device execution with DMA. In *Hardware and Software: Verification and Testing*, pages 236–251, Cham, 2014. Springer International Publishing.

Malte H Schwerhoff. *Advancing automated, permission-based program verification using symbolic execution*. PhD thesis, ETH Zurich, 2016.

Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 471–481, Seattle, Washington, USA, June 2013. ACM.

Konrad Slind and Michael Norrish. A brief overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32, Montréal, Canada, August 2008. Springer.

Yong Kiam Tan, Magnus Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29, February 2019.

Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. Krust: A formal executable semantics of rust. In *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 44–51. IEEE Computer Society, 2018. URL `https://doi.org/10.1109/TASE.2018.00014`.

Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. Oxide: The essence of Rust. *CoRR*, abs/1903.00982, 2019. URL `http://arxiv.org/abs/1903.00982`. Preprint.

Felix A Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C Pereira, and Peter Müller. Gobra: Modular specification and verification of Go programs. In *International Conference on Computer Aided Verification*, pages 367–379. Springer, 2021.

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in Coq. *Proceedings of the ACM on Programming Languages*, 4:51:1–51:32, January 2020.