SMT-based Deductive Verification of Device Drivers using the Pancake-to-Viper Transpiler

Samuel Tyler
samuel.tyler@unsw.edu.au
UNSW Sydney
Sydney, Australia

Alessandro Legnani*
alessandro@neutrality.ch
ETH Zürich
Zürich, Switzerland

Junming Zhao
junming.zhao@unsw.edu.au
UNSW Sydney
Sydney, Australia

Zhewen Shen†
zhewen.shen@unsw.edu.au
UNSW Sydney
Sydney, Australia

Rihui Wu
rihui.wu@unsw.edu.au
UNSW Sydney
Sydney, Australia

Miki Tanaka
miki.tanaka@unsw.edu.au
UNSW Sydney
Sydney, Australia

Gernot Heiser
gernot.heiser@unsw.edu.au
UNSW Sydney
Sydney, Australia

Abstract

We present comprehensive SMT-based deductive verification of a deployable, high-performance, real-world Ethernet driver written in the Pancake language for LionsOS. We develop a Viper frontend for Pancake that largely retains Viper's annotation syntax, and encode the driver's device and OS interfaces in Viper, resulting in an efficient and modular verification process.

CCS Concepts: • **Software and its engineering → Operating systems**; **Formal software verification**.

Keywords: Device driver verification, SMT solvers, Automated deductive verification

ACM Reference Format:
Samuel Tyler, Alessandro Legnani, Junming Zhao, Zhewen Shen, Rihui Wu, Miki Tanaka, and Gernot Heiser. 2026. SMT-based Deductive Verification of Device Drivers using the Pancake-to-Viper Transpiler. In *Workshop on Programming Languages and Operating Systems (PLOS '26), September 29–October 2, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3831586.3838153

Availability: Tools and artefacts from this work are available as open source from https://trustworthy.systems/software/plos26/.

*Now with Neutrality.
†Now with Arista.

PLOS '26, Prague, Czech Republic

ACM ISBN 979-8-4007-2877-8/26/09
https://doi.org/10.1145/3831586.3838153

1 Introduction

Device drivers are known to be among the most error-prone parts of an operating system (OS), accounting for the majority of CVEs reported for Linux in the period 2018–22 [Åman Pohjola et al. 2023]. This makes device drivers a high-priority target when verifying OS components.

A serious challenge is the commodity nature of peripheral devices: different platforms (SoCs or processor boards) generally feature different device implementations with different interfaces for the same peripheral class (e.g. Ethernet), requiring different drivers; and new platforms (with new devices) appear all the time. This implies that expensive approaches based on interactive theorem proving (ITP) are unlikely to solve the driver verification problem in practice. Instead we employ *automated deductive verification*, backed by SMT-solvers, in the hope that this will enable a degree of verification re-usability for other drivers of the same class.

A key enabler of our approach is Pancake [Åman Pohjola et al. 2023], a C-like language designed for low-level systems code. It has a verified compiler and a well defined and simple semantics, which simplifies reasoning about programs.

A second enabler is the device driver model of the modular seL4-based LionsOS [Heiser et al. 2025], which is radically simplified yet highly performant. LionsOS drivers are *single-threaded*, isolated, location-transparent event handlers, a model originally proposed for Linux [Ryzhyk et al. 2009a].

We develop a Pancake frontend ("transpiler") for Viper [Müller et al. 2016], which translates annotated Pancake programs into the Viper intermediate language (IL), which we then verify using the SMT-backed verifier. We specify the device interface as well as the driver's interface to the rest of the operating system (OS interface) in Viper, enabling automatic verification of the driver. We demonstrate

the approach by verifying a high-performance driver for a popular 1 Gb/s Ethernet network interface card (NIC).

We expect this approach to work not only for drivers, but also other LionsOS modules, which are of comparable complexity. In particular, this would prove that the rest of the OS adheres to the OS-side interface we specify for the driver. Our concurrent work on extracting device interface specifications from the device's Verilog source [Murphy et al. 2025] aims to provide provably correct specifications.

In summary, the contributions of our work are:

1. An automated deduction-based verification framework for OS modules, based on a Pancake-to-Viper transpiler (Section 2) that supports formal specifications using Viper-like annotations in the Pancake source code;
2. Viper formalisations of the device- and OS-side interface protocols of an Ethernet driver (Section 3.2.3);
3. proof that a Pancake-implemented driver adheres to the device and OS protocols, moves data correctly, and does not cause lock-up (Section 3.3);
4. evaluation demonstrating that the verified Pancake implementation of the driver performs within 20% of the optimised (unverified) original C version (Section 4.2), most of which will be eliminated by work in progress on the Pancake compiler.

2 Pancake-to-Viper Transpiler

Our transpiler translates annotated Pancake programs into Viper IL, similar to the frontend part of Viper-based verifiers such as Gobra [Wolf et al. 2021]; the developer writes specifications as source-code annotations. The resulting Viper IL is then verified using a Viper backend; we use Viper's symbolic-execution backend *Silicon* to discharge queries.

Our annotations are mostly standard Viper expressions, extended with Pancake-specific syntax for words, driver-local memory, and shared-memory accesses. This is sufficiently expressive (Section 3) yet efficient (Section 4.1) for driver verification, and we expect for OS modules more generally.

Internally, the transpiler first invokes a diagnostic mode of the Pancake compiler to obtain the parsed abstract syntax tree (AST), inclusive of the source-level annotations. It then emits Viper code representing Pancake control flow, variables, word values, memory, and logical assertions. Each Pancake function is emitted into its own Viper file.

During driver verification, the generated Viper code is checked together with external Viper model files, which define the Viper methods that encode the driver's interactions with the device and the rest of the OS (see Section 3.2.3).

We now describe the main translation choices.

2.1 Machine words

Pancake's only primitive type is the machine word, and its arithmetic has fixed-width wraparound semantics. We give two options for encoding machine words in Viper: either as `Int` values, or with a `BitVecDomain` interpreted as SMT-LIB fixed-size bitvectors [Barrett et al. 2010].

The user structures their code into functions that primarily use either integer or bitwise operations. This avoids mixed integer and bitvector reasoning in the same SMT context, which is difficult for SMT solvers to handle. In practice, this separation is not overly onerous. When bitwise operations occur in an `Int`-encoded function, the transpiler converts the relevant words to bitvector and then back to integer.

Arithmetic is encoded to three-address form (single-operation assignments such as `t=a+b`). When using Viper `Int` encoding, the transpiler emits a bounds check after each arithmetic operation. This does not affect our driver's verification, as the driver does not rely on wraparound semantics.

2.2 Local memory

Viper uses *permission* predicates to declare which memory regions and global variables code may access, based on implicit dynamic frames (Section 3.2.1). Driver-local heap memory is untyped and word-addressed, and is accessed by load and store instructions; Pancake disallows pointers to the stack. We can thus represent local heap memory in Viper as a single word-addressed memory abstraction. Each global variable is represented as a named global resource. Stack variables are translated directly to Viper variables, which do not use permission predicates since they only exist in local scope.

2.3 Shared memory

The encoding of driver-local memory applies only to memory whose contents are controlled exclusively by the driver. Pancake uses separate shared load and store operations for memory regions that may be concurrently updated by the device or the OS; this includes memory-mapped (MMIO) device registers, direct memory access (DMA) descriptors, and state shared with other OS components. These operations are similar to C `volatile` accesses: a read may observe a value not produced by any previous driver write, meaning that encoding such regions as ordinary driver-local memory would be incorrect. In addition, these operations must not be reordered, fused, introduced or removed by the compiler.

For these operations, the transpiler emits a call to a Viper method supplied by an external model file. Top-level Pancake annotations map each region of shared memory to the corresponding external Viper methods. The method specification gives the contract of the shared-memory operation: at the generated call site, Viper checks that the driver establishes the precondition, and then makes the postcondition available to the rest of the proof. Section 3.2.3 explains how these contracts are written for device and OS interface interactions.

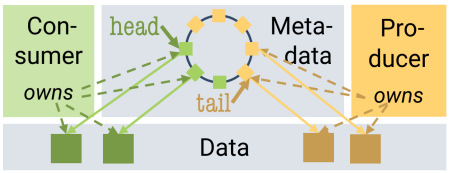

Figure 1. sDDF memory regions and queue ownership.

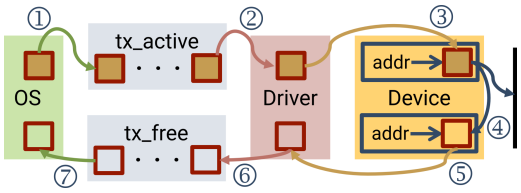

Figure 2. Movement of a Tx buffer.

3 Verification

3.1 Target

We target an Ethernet NIC used on many embedded Arm platforms. It is controlled via MMIO registers, as well as circular transmit (Tx) and receive (Rx) queues ("ring buffers") holding descriptors pointing to data buffers passed between the OS and device (accessed by DMA). These descriptors consist of the `stat` field, containing control bits, the `addr` field, pointing to a data buffer in DMA memory, and the `len` field, containing the length of the data buffer. The device protocol uses the `stat` field to indicate which party (device or driver) holds exclusive control of ("owns") each descriptor and associated buffer. The MMIO registers and descriptors are mapped uncached by the OS (Device-nGnRnE in Arm), so the driver only needs to ensure accesses are properly ordered by using appropriate memory barriers. Data buffers are mapped cached (Normal write-back in Arm), and software cache management is used in addition.

The design of the *seL4 device driver framework* (sDDF) used by LionsOS emphasises strict separation of concerns: a driver's sole purpose is to abstract a device-specific hardware interface into a device-independent but device-class specific OS interface, meaning the OS interface of all Ethernet drivers looks the same (and cache management is performed by a module separate from the driver). This OS interface uses a set of zero-copy, lockless, bounded, single-producer, single-consumer (SPSC) queues, allocated in the *metadata* region shared between the driver and the OS (see Figure 1).

A queue is implemented as a circular array of pointers to data buffers in the *data region* shared between OS and device. Each queue has a *head* index, where the consumer dequeues buffers, and a *tail* index, where the producer enqueues. Notably, the consumer is the sole writer to the head, and the producer is the sole writer to the tail. The producer and consumer may or may not share a processor core.

The OS interface has four queues: the `rx_active` queue transfers buffers filled with incoming data to the OS (driver is producer and OS is consumer) while the `rx_free` queue transfers buffers back to the driver to hand them to the device for receiving more data (inverted producer and consumer). Outgoing data is transferred though a corresponding `tx_active`, `tx_free` queue pair. Queue entries have well-defined *owners*: the consumer owns all entries between the head (inclusive) and the tail (exclusive), while the producer owns the rest, as shown in Figure 1.

Figure 2 shows the journey of a Tx buffer: The OS fills a buffer with output data, and ① enqueues it into the `tx_active` queue, from where ② it is dequeued by the driver. The driver then ③ places the buffer into an appropriate descriptor of the device and alerts the device according to the device protocol. The device ④ transmits the buffer's data onto the wire, then ⑤ hands the descriptor and buffer back to the driver and raises an interrupt. The driver ⑥ inserts the buffer back into `tx_free` from where ⑦ the OS regains it. Rx buffers travel in reverse through their respective queues. Software components use an asynchronous signal to alert each other of pending work.

Each buffer has at any time an exclusive owner (OS, driver or device). Violating the ownership protocol may lead to buffers being lost or overwritten (and the verification found such a case). Note that the driver only handles references to data buffers, not the data itself, and therefore does not have the data region mapped in its address space.

3.2 Formalism

To reason about the driver's interactions, we need a formal model (the *interface model*) of the shared-memory interface between driver and OS. The transpiler connects the generated Viper code of the driver to the interface model through calls to external Viper methods that model shared memory accesses (Section 2.3).

These methods do not need to model the concrete implementation of the device or OS, only the *contract* governing their interactions with the driver, i.e., preconditions and postconditions constituting the specification of the shared memory access. Shared state is modelled by fields provided to every method. As indicated in Section 2.3, shared-memory accesses are transpiled to a call to a Viper method, and the preconditions are verified and the postconditions assumed for the rest of the proof. Load postconditions describe observations returned to the driver, while store postconditions describe updates committed by the driver.

3.2.1 Use of implicit dynamic frames Viper is built upon *implicit dynamic frames* (IDF) [Smans et al. 2012], a variant of separation logic. It uses *permission predicates* to indicate access rights to a resource, from 0 (none) to 1 (read and write), which are linked to fields and passed around the program at method calls. Thus, IDF provide a convenient

logic for capturing the notion of ownership explained in Section 3.1: We represent ownership as holding full permission, and passing ownership as a permission change.

Another useful property of IDF is *automatic framing*. When the caller of a method passes less than the entire permission it holds to a field to the callee, Viper infers that the value of that field did not change. This is extremely convenient, as most functions only touch a small subset of the interface model – we get unchangedness properties for free. These two features reduce the mental load for developers.

3.2.2 Concurrency Modelling the interface *contract* rather than a concrete implementation allows us to address concurrency in a way that abstracts over the interleaving execution of the driver, device and OS.

Our formalism does not include any notion of global time. As such, the shared state fields do not claim to model a *current state* of the queues; they model the most up-to-date understanding of the state *from the driver's perspective*. For example, the tail of a queue for which the driver is the consumer may have been incremented, but because the driver has not yet read that memory, its local view of the state remains unchanged; i.e. will not have the tail incremented.

Then, how do we update the model with the actual external state? The only way the driver gains information about the external state is *through shared memory loads*. So, the postconditions on shared memory loads encode a *non-deterministic relation* between the previous view of external state and a refined, more recent view of the external state (albeit not necessarily entirely current). This relation makes no assumption about the amount of time that has passed since the previous view of external state was current. Thus, load methods usually have non-deterministic return values.

Since the model encodes the most up-to-date view of external state based on the information the driver has, preconditions *anywhere* must assume that the state in the model is arbitrarily outdated compared to the actual external state.

However, all of our preconditions are *stable properties*, which will not become false under unbounded execution of the device or OS. For example, a precondition stating that the difference between a queue's tail and head does not exceed its capacity is stable; the OS will never make this property false. Thus, we can uphold the assumption that the state in the model is arbitrarily outdated.

This approach can capture the requirements of release-acquire based weak memory models, by moving the application of the non-deterministic relation to the location of the memory barriers, similar to that of fenced separation logic [Doko and Vafeiadis 2016]. We use our own encoding for weak memory that we aim to prove to correspond to the underlying hardware concurrency model.

3.2.3 Interface model encoding Figure 3 gives an example of the use of pre- and postconditions on shared-memory store methods; the preconditions require that the program

```
1  method store_tx_free_tail(gv: Ref, value: Int)
2    // preconditions on the nature of the inputs
3    requires acc(gv.tx_free.net_tail)
4    requires value == (gv.tx_free.net_tail + 1)
5                      % (MAX_INT16 + 1)
6    // preconditions on the state
7    requires gv.tx_free.net_tail - gv.tx_free.net_head
8             != QUEUE_CAPACITY
9    // (deterministic) postconditions on the state
10   ensures gv.tx_free.net_tail ==
11            old(gv.tx_free.net_tail) + 1
```

Figure 3. Example (truncated) store method, for `tx_free` shared tail metadata.

```
1  method load_tx_free_head(gv: Ref, address: Int)
2    returns (retval: Int)
3    // take entire permission (allows non-determinism)
4    requires acc(gv.tx_free.net_head)
5    ensures acc(gv.tx_free.net_head)
6    // updating the state (encodes non-determinism)
7    ensures gv.tx_free.net_head >=
8      old(gv.tx_free.net_head)
9    ensures gv.tx_free.net_tail >= gv.tx_free.net_head
10   // postconditions on the returned value
11   ensures retval == gv.tx_free.net_head %
12     (MAX_INT16 + 1)
```

Figure 4. Example (truncated) load method, for `tx_free` head field, including non-determinism.

has access to the queue tail (Line 3), that the tail can only be incremented by one (Lines 4–5), and that the queue is not full (Lines 7–8). The postcondition says that the method also updates the interface model with the change (Lines 10–11).

Similarly in Figure 4, load methods have postconditions that encode updates to the view of external state based on information gained (the aforementioned *non-deterministic relation*). We utilise a convenient Viper feature: when automatic framing (Section 3.2.1) does not kick in, the field is given a non-deterministic value. Load methods always take the entire permission held by callers to definitively avoid automatic framing: here, the entire permission to `tx_free.net_head` is taken (Line 4), so that the caller places a non-deterministic value in `tx_free.net_head`, which is constrained by the relation in the postconditions. Semantically, these postconditions say that the OS may have incremented the head of the queue an arbitrary number of times, and the head did not go past the tail. The last postcondition relates the returned value to the interface model: the returned value must be the model's value of the head of the queue, wrapped to 16 bits.

On a platform with weak memory, the updates to the interface model, such as lines 10–11 of Figure 3 and lines 7–10 of Figure 4, are added to Viper's proof context not at the method call site, but at memory barriers, reflecting the guarantees on the visibility of memory operations.

Encoding interfaces supports modularity of verification, by verifying against contracts irrespective of how they are implemented on the other side of the interface.

Interface	Viper
Device	549
OS	443

Program	C	Unverified Pancake	Verified Pancake
Driver	159	295	362
Queues	116	60	64

Table 1. Verification target statistics (LOC).

3.2.4 Tracking history We need to reason about buffers which, in the past, we have added or removed from the device and queues, including those no longer in the shared state within the interface model. Any shared memory access that removes or adds a buffer from or to the device or a queue therefore records that buffer as removed or added in a ghost set,[1] which represents the history of the driver's operation. Changes to ghost sets are represented in shared memory method postconditions. This simple approach is efficient yet sufficiently powerful.

3.3 Properties verified

We prove four safety and functional properties for our device driver. The first two do not have convenient associated theorems, but are encoded in pre- and postconditions on shared memory operations and ensured by the program's verification against calls to shared memory operations. The remaining two properties are captured in explicit postconditions on the ghost sets of Section 3.2.4 at driver entrypoints.

3.3.1 Device protocol adherence We formalise the device protocol by translating the hardware manual's prose into shared memory pre- and postconditions. This is error-prone, as the manuals tend to be imprecise and frequently wrong [Ryzhyk et al. 2009a] – we are addressing this through verified device-interface specifications [Murphy et al. 2025].

3.3.2 Queue integrity We prove that the driver only touches the head of a queue it consumes, or the tail of a queue it produces, and that it does not overflow or underflow a queue, nor read from an empty queue. We prove that the driver honours the ownership contracts of the queues and device metadata (Section 3.1), which are encoded in pre- and postconditions as gaining and losing field permissions (Section 3.2.1). We also carry global invariants for well-formedness of the shared state with the device and queues.

3.3.3 Correct buffer movement The driver is proved to adhere to correct buffer movement operation (Section 3.2). Given that every buffer is only ever owned by either the device, driver or OS, for the driver it is sufficient to show that every buffer removed from the `tx_active` queue ends up handed off to the device; and every buffer received as the result of an interrupt is placed into the `tx_free` queue; equivalent properties hold for the Rx side. The driver is proved not to de-reference buffer pointers (which would lead to a memory fault, see Section 3.1).

[1] A ghost set contains data that does not actually exist within the program.

Property	Time (pw)	Annotations (LOC)
Device Protocol Adherence	2	228
Queue Integrity	1.5	123
Buffer Movement	4	350
Completion Condition	2.5	114
(inlining duplication)	-	250
(weak memory)	1.5	280
(other)	-	156
Total	11.5	1501

Table 2. Verification cost statistics.

3.3.4 Completion condition We prove that the driver will process available buffers. As the device does not guarantee processing packets in FIFO order, the driver cannot enqueue all currently available packets without scanning the whole queue (which would be inefficient). Instead, we guarantee that if the `tx_active` queue is non-empty and the device's Tx queue is empty, the driver will enqueue a buffer to the device – this condition will eventually be true under correct operation of the device. Equivalent conditions are proved for the reverse direction and for Rx queues.

The combination of these four properties establish the driver's safety and correctness – to the best of our knowledge the most comprehensive verification of a non-toy device driver to date. Their specification is sufficiently abstract to not unduly impact verification efficiency.

4 Evaluation

We verify two LionsOS components: the NIC driver and the library implementing the SPSC queues (Section 3.1). We prove all properties discussed in Section 3.3, without assuming that the driver executes on the same core as the rest of the OS, i.e. the system is multicore-safe. The verification was effective: it exposed one queue contract violation and two weak-memory bugs, all of which were rather subtle errors.

4.1 Verification effort

To ease verification we made two minor modifications to the driver. 1. *Splitting Functions:* We extract some code segments into inlined functions. This reduces the complexity of the SMT solver's proof obligations and reduces verification time (overall and per-function), without affecting performance, at the cost of significant duplication of annotations. 2. *Use of Global Variables:* We originally avoided global variables for performance reasons (Section 4.2). However, this makes the program more difficult to reason about, so we reverted to global variables, which are simpler to encode (see Section 2.3). Table 1 shows the lines of code in the program and the interface model. The Pancake queues change data layout compared to C to eliminate function duplication.

Once the interface protocols were encoded (Section 3.2.3), the verification (i.e. annotating) was mostly straightforward,

performed by an undergraduate student (first author) without prior expertise in automated program verification (but see Section 5 for some of the challenges we encountered). Table 2 shows the annotation overhead. Parentheses indicate the annotations are not tied to any specific property; "other" represents common definitions that link the annotations with the interface model. We perform simple macro expansion and concatenate files before handing them to the transpiler.

Verification of most functions takes seconds to a few minutes on a modern desktop. Four more complex functions take 15–25 minutes; and the four most involved functions take 1–3 hours to verify.

As we prove properties, we refine the interface models to add detail and capture additional behaviour as required. Owing to the sDDF's strict separation of concerns, encoding the OS interface in pre/postconditions is straightforward, demonstrating the modularity benefits of the encoding of Section 3.2.3. Much of this should be readily re-usable for other Ethernet drivers, because all Ethernet drivers share the same OS interface. Furthermore, as sDDF network-device classes have much commonality, there will likely be significant opportunity for reuse with similar devices (e.g. busses).

The similarity of the Rx and Tx paths of the driver means that having annotated one path, the annotations can serve as a template for the other path, including copying some parts verbatim, with the caveat of largely replicated annotations. Existing abstractions are insufficient to avoid replication.

4.2 Performance

We evaluate performance on an Avnet MaaXBoard with a NXP i.MX8MQ SoC (four Arm Cortex-A53 cores, 2 GiB of RAM, and a 1 Gb/s on-chip NIC), clocked at a fixed 1 GHz. A separate system sends data at a specified rate, using 1.5 kB Ethernet frames, which the test system receives and echoes back. We measure returned throughput, latency (RTT), and driver as well as whole-system CPU utilisation.

Our original ("unverified") Pancake driver was translated from the LionsOS C driver. Presently, Pancake-generated code issues global load-store sequences for all global-variable accesses, which is inefficient if the same variable is used multiple times. The original Pancake driver avoided this inefficiency by caching global variables in locals. However, this complexity increases verification effort, so our *verified* Pancake is a more direct translation of the original C.

All three drivers easily handle the load, with the unverified Pancake driver adding 0–10% CPU over C, and the verified driver adding 10–20% (latency changes are well within a standard deviation). As a fraction of the system's total CPU usage, the overheads are in the noise, but would be (somewhat) noticeable if the whole system was implemented in Pancake. These overheads will be all but eliminated by current work on improving Pancake's handling of globals.

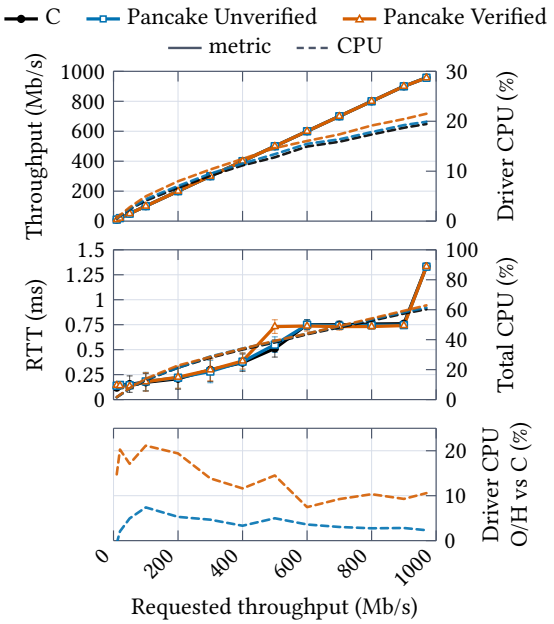

Figure 5. Performance of the verified driver against the C and unverified Pancake baselines.

5 Challenges with Viper and SMT solvers

Many of our challenges are typical to deductive program verification, but there are Viper-specific ones.

While Viper proved powerful, some of the functions took it to the limits, transpiling to over 2000 lines of Viper IL, and using many advanced Viper features. Consequently we often found ourselves waiting for the Viper backend to complete verification, particularly in later parts of the verification process when dealing with the more complex functions.

We found Viper's permission model challenging. Holding access permission greater than 1 is considered an inconsistent assumption and implies false. However, with many lines of assumptions it is easy to accidentally introduce too much permission, especially when they are hidden behind functions or predicates. We avoid this foot gun by adding `refute false` statements, which raise an error if false can be derived. We plan to side-step this issue by adding transpiler features for automating access permission value calculation, which should significantly reduce developer pain.

Quantified properties can be frustrating. The restrictive trigger format meant that at times expressing a trigger in the most useful way was not possible. We also found that more advanced language features interacted with quantifier triggers in unexpected and bizarre ways. In some cases this forced the use of a crude form of manual instantiation of quantifiers by specifying dummy triggers.

We experienced the typical "hand-holding" process when the SMT solver was unable to prove a property. SMT timeouts required significant, order-of-magnitude adjustments, and appropriate timeouts vary between Viper files. Viper's

default SMT solver Z3 [de Moura and Bjørner 2008] failed on many queries, while the cvc5 solver [Barbosa et al. 2022] can cope more often. However, for this specific application, cvc5 is generally slower. We note that recent SMT research, such as portfolio solvers [Barrett et al. 2024] or the conjecture of Cebeci et al. [2025], offer hope for improvement.

6 Related Work

Static analysis/model-checking work (e.g. SLAM [Ball and Rajamani 2002]) has been used to establish simple safety properties on drivers. The Verisoft project proved functional correctness of a simple ATAPI hard disk device driver of about 30 lines of assembly using ITP [Alkassar 2009; Alkassar and Hillebrand 2008]. Penninckx et al. [2012] use the SMT-based VeriFast [Jacobs et al. 2010] to prove basic memory and data-race safety of a Linux USB keyboard driver, as well as correct use of the Linux USB API, but do not tackle OS-protocol adherence nor functional correctness. Bierhoff and Hawblitzel [2007] use Spec# [Barnett et al. 2005] to verify correct communication of a network interface driver with the device, including device setup, but do not reason about the OS interface, and run on x86 thus avoiding weak memory concerns. More recently, Chen et al. [2024] explored using Verus [Lattuada et al. 2023] to verify device drivers written in Rust, but have not yet verified a real-world driver. CertiKOS [Chen et al. 2016], Erbsen et al. [2021] and Ironclad [Hawblitzel et al. 2014], while mentioning functional verification of device drivers integrated into wider systems, do not focus on driver verification, do not provide real-world driver examples, and do not evaluate driver performance. Pereira et al. [2025] verify safety and functional correctness of network protocol implementations using Gobra [Wolf et al. 2021]. While not verifying drivers, they report code annotation effort and verification cost in line with our work.

Summers and Müller [2020] encode various program logics for weak-memory programs in Viper. These are restricted to single-location invariants, are not expressive enough for our driver, and the frontend language is incompatible with other Viper programs. We therefore currently specify the memory model with Pancake annotations.

Ryzhyk et al. [2007] introduced the Tingu formalism for describing device protocols, which they subsequently used in the Dingo framework that simplified drivers for both Linux and the OKL4 microkernel though an event-driven active-driver model [Ryzhyk et al. 2009a]; our driver model is based on their idea. Termite then used this approach for specifying both device and OS interfaces, and automatically synthesised drivers from these interface specifications [Ryzhyk et al. 2009b]. A similar synthesis workflow has recently been proposed with Ghost Writer [Wang et al. 2023], using behaviour trees. Termite driver performance was close to manually-written code (although with significant code bloat), while the performance of Ghost Writer's drivers are unclear. Note that while synthesis can avoid many driver bugs, it does not guarantee correctness, and it makes enforcement of some properties difficult, e.g. concurrency safety in Ghost Writer.

7 Conclusions and Further Work

We demonstrate that full, efficient, modular verification of performant, real-world device drivers is possible with automated deductive verification. We demonstrate this on an Ethernet driver implemented in Pancake, using a transpiler to Viper that allows annotations on the Pancake source code. We develop expressive formal models of the driver's hardware and OS interfaces, suitable for multicore processors.

The transpiler is presently written in Rust and part of the trusted computing base. We are currently rewriting the transpiler in the HOL4 theorem prover and are developing an appropriate semantics for Viper to enable proof that the emitted Viper IL corresponds to the original annotations. We are exploring choices for stronger backend correctness guarantees, such as Viper's (presently outdated) correctness story for Carbon [Parthasarathy 2024].

As noted earlier, the weak memory model is currently manually encoded into the Pancake code annotations and Viper model files. We plan improvements to the transpiler to partially automate this, and further experiments with weak memory models and proof techniques.

There are some issues that currently inhibit broader use of our approach, especially by engineers without a formal-methods background. Our handling of mixed integer, bitwise and logical operations (Section 2.1) is suboptimal and we plan to establish a more integrated and efficient approach. Annotation duplication arising from similar algorithms (Section 4.1) and a lack of data structure abstraction should be addressed with further automation; e.g. an annotation templating feature that is more powerful than Viper's abstractions.

Integrating interactive proofs for logically complicated properties instead of relying solely on automated verification might be a way around some of the challenges we experienced. Such interactive proofs would make use of properties proved by automated deduction, rather than dumping out a failed proof obligation, as done in some existing work [Gladshtein et al. 2026]. Why3 [Filliâtre and Paskevich 2013] may provide useful insights. We also note separation logic may not be ideal for some other OS modules, suggesting possible addition of non-Viper backend options.

Acknowledgments

This work was performed under the PISTIs-V project, funded by the German agency *Agentur für Innovation in der Cybersicherheit GmbH* (Cyberagentur) under the *Ecosystem Formally Verifiable IT – Provable Cybersecurity* (EVIT) Program. We thank Toby Murray of the University of Melbourne for hosting Alessandro Legnani for his work on this project.

References

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

Eyad Alkassar and Mark A. Hillebrand. 2008. Formal Functional Verification of Device Drivers. In *Verified Software: Theories, Tools and Experiments (Lecture Notes in Computer Science)*, Vol. 5295. Springer, Toronto, Canada, 225–239.

Johannes Åman Pohjola, Hira Taqdees Syeda, Miki Tanaka, Krishnan Winter, Tsun Wang Sau, Benjamin Nott, Tiana Tsang Ung, Craig McLaughlin, Remy Seassau, Magnus O. Myreen, Michael Norrish, and Gernot Heiser. 2023. Pancake: Verified Systems Programming Made Sweeter. In *Workshop on Programming Languages and Operating Systems (PLOS)*. ACM, Koblenz, DE, 1–9.

Thomas Ball and Sriram K. Rajamani. 2002. The SLAM project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. Association for Computing Machinery, New York, NY, USA, 1–3. https://doi.org/10.1145/503272.503274

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science)*, Vol. 13243. Springer, Munich, Germany, 415–442.

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. 2005. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Springer Berlin Heidelberg, Nice, France, 49–69.

Clark Barrett, Pei-Wei Chen, Byron Cook, Bruno Dutertre, Robert Jones, Nham Le, Andrew Reynolds, Kunal Sheth, Chriss Stephens, and Mike Whalen. 2024. SMT-D: New strategies for portfolio-based SMT solving. In *Conference on Formal Methods in Computer-Aided Design*. IEEE, Prague, Czech Republic, 1–10.

Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In *Proc. 8th Int. Workshop on Satisfiability Modulo Theories*. Edinburgh, UK, 1–85.

Kevin Bierhoff and Chris Hawblitzel. 2007. Checking the Hardware-Software Interface in Spec#. In *Workshop on Programming Languages and Operating Systems (PLOS)*. ACM, Stevenson, WA, US, 5.

Can Cebeci, Nikolaj Bjørner, George Candea, and Clément Pit-Claudel. 2025. A Conjecture Regarding SMT Instability. In *International Workshop On Satisfiability Modulo Theories*. CEUR-WS, Glasgow, UK, 136–147.

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

Xiangdong Chen, Zhaofeng Li, Jerry Zhang, and Anton Burtsev. 2024. Veld: Verified Linux Drivers. In *Workshop on Kernel Isolation, Safety and Verification*. ACM, Austin, TX, USA, 23–30.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science)*, Vol. 4963. Springer, Budapest, Hungary, 337–340.

Marko Doko and Viktor Vafeiadis. 2016. A Program Logic for C11 Memory Fences. In *International Conference on Verification, Model Checking and Abstract Interpretation*. Springer Berlin Heidelberg, St. Petersburg, FL, USA, 413–430.

Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration Verification across Software and Hardware for a Simple Embedded System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, Virtual, 604–619.

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *European Symposium on Programming*. Springer Berlin Heidelberg, Rome, Italy, 125–128.

Vladimir Gladshtein, George Pîrlea, Qiyuan Zhao, Vitaly Kurin, and Ilya Sergey. 2026. Foundational Multi-Modal Program Verifiers. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Vol. 10. ACM, Rennes, France, 32. https://doi.org/10.1145/3776719

Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX, Broomfield, CO, US, 165–181. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel

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. 2025. Fast, Secure, Adaptable: LionsOS Design, Implementation and Performance. *arXiv preprint* (Jan. 2025), 14. https://arxiv.org/abs/2501.06234

Bart Jacobs, Jan Smans, and Frank Piessens. 2010. A Quick Tour of the VeriFast Program Verifier. In *Asian Symposium on Programming Languages and Systems (APLAS) (Lecture Notes in Computer Science)*, Vol. 6461. Springer, Shanghai, China, 304–311.

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, Cascais, Portugal, 30. https://doi.org/10.1145/3586037

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

Liam Murphy, Albert Rizaldi, Lesley Rossouw, Chen George, James Treloar, Hammond Pearce, Miki Tanaka, and Gernot Heiser. 2025. High-Fidelity Specification of Real-World Devices. In *Workshop on Programming Languages and Operating Systems (PLOS)*. ACM, Seoul, Republic of Korea, 60–67.

Gaurav Parthasarathy. 2024. *Formally Validating Translational Program Verifiers.* Ph.D. Dissertation. ETH Zurich.

Willem Penninckx, Jan Tobias Mühlberg, Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Sound Formal Verification of Linux's USB BP Keyboard Driver. In *NASA Formal Methods Symposium (Lecture Notes in Computer Science)*, Vol. 7226. Springer Berlin Heidelberg, Norfolk, VA, USA, 210–215.

João Pereira, Tobias Klenze, Sofia Giampietro, Markus Limbeck, Dionysios Spiliopoulos, Felix Wolf, Marco Eilers, Christoph Sprenger, David Basin, Peter Müller, and Adrian Perrig. 2025. Protocols to Code: Formal Verification of a Secure Next-Generation Internet Router. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25)*. ACM, Taipei, Taiwan, 1469–1483. https://doi.org/10.1145/3719027.3765104

Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. 2009a. Dingo: Taming Device Drivers. In *EuroSys Conference*. ACM, Nuremberg, DE, 275–288.

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

Leonid Ryzhyk, Ihor Kuz, and Gernot Heiser. 2007. Formalising device driver interfaces. In *Workshop on Programming Languages and Operating Systems (PLOS)*. ACM, Stevenson, WA, USA, 5.

Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit dynamic frames. *ACM Transactions on Programming Languages and Systems* 34 (May 2012), 58.

Alexander J. Summers and Peter Müller. 2020. Automating deductive verification for weak-memory programs (extended version). *International Journal on Software Tools for Technology Transfer* 22, 6 (2020), 709–728.

Bingyao Wang, Sepehr Noorafshan, Reto Achermann, and Margo Seltzer. 2023. Synthesizing Device Drivers with Ghost Writer. In *Workshop on Programming Languages and Operating Systems (PLOS) (PLOS '23)*. ACM, Koblenz, Germany, 10–17. https://doi.org/10.1145/3623759.3624545

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