A Formally Verified NAT Stack

Solal Pirelli
EPFL, Switzerland

Arseniy Zaostrovnykh
EPFL, Switzerland

George Candea
EPFL, Switzerland

Abstract

Prior work proved a stateful NAT network function to be semantically correct, crash-free, and memory safe [29]. Their toolchain verifies the network function code while assuming the underlying kernel-bypass framework, drivers, operating system, and hardware to be correct. We extend the toolchain to verify the kernel-bypass framework and a NIC driver in the context of the NAT. We uncover bugs in both the framework and the driver. Our code is publicly available [28].

CCS Concepts

• **Software and its engineering → Formal software verification** • Networks → Middle boxes / network appliances

Keywords

Network Functions; Software Verification; Kernel Bypass

ACM Reference format:

Solal Pirelli, Arseniy Zaostrovnykh and George Candea. 2018. A Formally Verified NAT Stack. In *Proceedings of KBNets '18, Budapest, Hungary, August 20, 2018*, 7 pages.
DOI: https://doi.org/10.1145/3229538.3229540

1 Introduction

Implementing network functions (NFs) in software offers more flexibility at a lower cost than purpose-built hardware. Frameworks such as the Data Plane Development Kit (DPDK) [5] have recently lowered the performance overhead of software NFs by bypassing the operating system kernel.

Zaostrovnykh et al. presented a formally verified Network Address Translator (NAT) [29]. They prove the NAT to be semantically correct, crash-free, and memory safe using a new toolchain called Vigor. They implement the NF in C using DPDK on Linux. We summarize their proof technique in §2.

Their proof applies to the application code and a library, totaling 2K lines of code, but not the modules below. As Figure 1 shows, the NAT code and library are verified, while the other modules are merely assumed to work correctly. This is not at all unusual – most verification work assumes the underlying modules to be correct. The problem is that the unverified modules may have bugs that affect the NAT's behavior.

KBNets '18, August 20, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5909-2/18/08...$15.00
https://doi.org/10.1145/3229538.3229540

Our main observation is that while DPDK and its network interface controller (NIC) drivers, as employed by the NAT, are large codebases (~90K lines of code), only a small fraction of their code needs to be verified in order to prove correctness of the entire NAT stack. This allows us to include the DPDK framework and a NIC driver in the verification. The result is a reduction of the trusted code base, as Figure 2 shows.

We model less modules from the stack and verify more. Instead of modeling DPDK, we model the C library and the hardware. This removes two large modules from the trusted code base. We believe that our technique can generalize. We discuss the background and our proof technique in §3.

During the proof effort, we discovered bugs in both DPDK and the NIC driver. These findings confirm the risks that software NFs take when assuming that the underlying modules are correct. We present and discuss these bugs in §4.

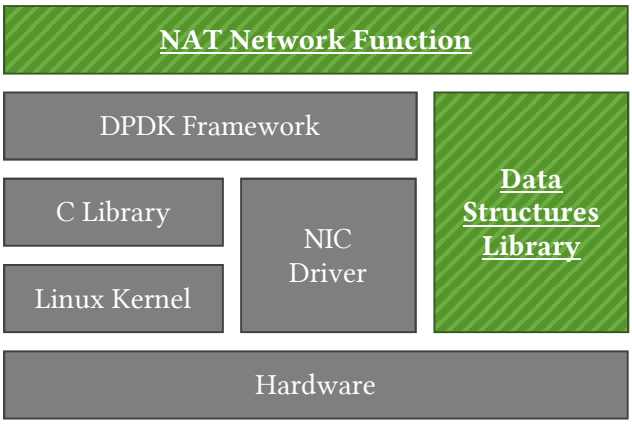

Figure 1: State of the NAT stack [29] **before our work. Only hashed green modules are proven to not cause the NAT to diverge from its specification.**

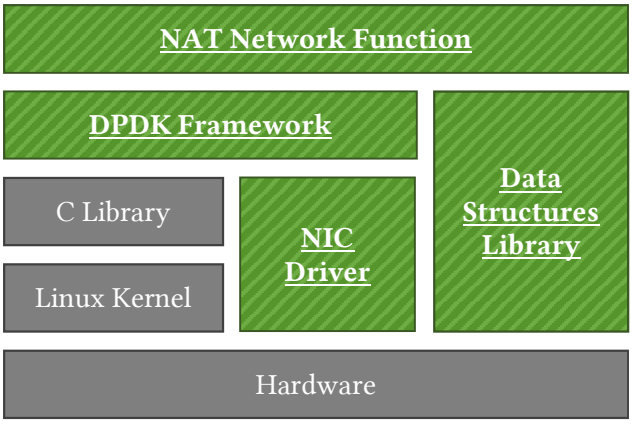

Figure 2: Verification state of the NAT after our work. (Diagram semantics are the same as in Figure 1)

We patch DPDK to fix some bugs and make verification easier, but this does not affect the performance of the NAT. We evaluate the resulting NAT end-to-end in §5.

We present limitations and ideas for future work in §6, discuss related work in §7, and conclude in §8.

2 Proving NAT Correctness with Vigor

The NAT presented in [29] was the first stateful NF proven to be semantically correct, using a toolchain called Vigor. The proof not only shows that the NAT will never crash but also that it always forwards or drops packets according to a formalized version of RFC 3022 [24]. We summarize the Vigor approach in this section.

The proof of correctness of the NAT combines theorem proving for data structures code (stateful) and symbolic execution for the rest of the code (stateless). Even though theorem proving requires a lot of human effort, NF developers can reuse the verified data structures for many NFs. Symbolic execution is automated, and easy to run on a new NF.

During symbolic execution of the NF, Vigor replaces the data structures code and the DPDK framework with models that return symbolic data. For instance, Vigor models DPDK's packet-receive function by a function that returns either one or no packet. Symbolic execution explores both possibilities. The packet itself, if there is one, has symbolic contents as well. If the NF has different behavior depending on a property of the packet, such as whether it is UDP or TCP, symbolic execution explores both options. The symbolic execution engine tracks constraints when exploring each possibility to remember the choices that it made, such as "there is a packet, and it is UDP".

The main loop of an NF is infinite and thus cannot be verified by merely executing it symbolically, since this would not terminate. To verify the loop, Vigor uses havocing [2, 29]. The symbolic execution engine remembers which sections of memory the NF writes to during a loop iteration, and replaces them with symbolic values in the next loop iteration. This continues until the engine finds a fixed point (i.e., further iterations do not expand the amount of symbolic memory), at which point it knows exactly which parts of memory can change during an iteration. Vigor proves that every iteration is correct by the following logic: if all memory that can change is symbolic, the symbolic execution engine explores all paths that result from iterating through the loop. If all such paths are correct, then all possible iterations are correct.

After running symbolic execution, Vigor verifies that the NF uses the data structures correctly, and that the models are correct approximations of the data structures code. This ensures that in the context of the NF, the models exhibit all possible behaviors of the real code.

The Vigor toolchain relies on VeriFast [17] for theorem proving, a modified version of KLEE [4] for symbolic execution, and a custom "validator" to stitch the two together.

3 Extending the Proof

Our objective is an NF that is formally proven to conform to its specification. A bug at any level of the stack may compromise this objective.

Since we cannot verify the entire stack at once, we must prioritize the choice of code to verify. We use two axes: code that is immature (and thus likely to have bugs) and code that has high potential for reuse in future NFs.

While hardware can also have bugs, we do not consider it a priority for NF verification. We only use basic packet send/receive features, not complex NIC features such as packet filtering or checksum offloading. While hardware may have undiscovered bugs, they are unlikely to be in the basic features.

Unlike the C library and the Linux kernel, which are mature and "battle-tested", DPDK is a large body of new code. Intel released the first public version, 1.8, in December 2014. New versions are released every few months [22]. Thus, we hypothesize that DPDK and its drivers are likely to have bugs.

Another reason to verify DPDK and its driver is reusability. Our approach generalizes to any network function that follows the design advocated in [29]: use the verified data structures, implement the Vigor NF interface (one method that takes in a packet and returns a forwarding decision), and avoid pointer arithmetic or similar "dangerous" C constructs.

Vigor employs exhaustive symbolic execution to verify stateless code, and theorem proving to verify data structures. We cannot use exhaustive symbolic execution to verify all the remaining unverified parts of the NAT stack because of the traditional problem of path explosion [3]. Faced with an exponential number of paths (or worse), exhaustive symbolic execution may not terminate in reasonable time.

The other tool used by Vigor, VeriFast [17], requires lots of annotations in the code. The libVig library [29] contains about 10 lines of annotations (pre-conditions, post-conditions and proofs) for each line of code. Using VeriFast to verify DPDK and its NIC drivers would require a lot of human effort. Furthermore, much of the work would have to be redone every time a new version of DPDK is released.

Our observation is that, while kernel-bypass frameworks such as DPDK may be *large*, writing a basic NF requires only a small subset of DPDK: initialization, receiving packets, and sending packets. This subset contains a lot of code, but it has simple control flow and almost all of it is simple operations such as reading or writing to device registers. There are few branches in the framework itself; most of those branches cause the execution to terminate early, such as not being able to initialize some subsystem. Thus, the symbolic execution engine does not encounter the path explosion problem at scale. Some parts of the framework may be difficult to symbolically execute, such as data structures with pointers, but those parts are not necessary to write an NF. Thus, we hypothesize that it is possible to symbolically execute the DPDK framework as used by NFs, with minor potential modifications.

The way we use symbolic execution implies that we only analyze code that is reachable in the NF under verification. Our approach therefore does not prove DPDK to be correct in an absolute sense. We only prove that the parts of DPDK used in the NF do not cause the NF to violate its specification.

We present the fundamental challenges we encountered, along with the insights that underlie our solutions, in §3.1. We present our implementation of the solutions for DPDK in §3.2, for the NIC driver in §3.3, and for the C library in §3.4.

3.1 Challenges

We now describe the challenges that we consider to be inherent to any proof of correctness of a kernel-bypass framework such as DPDK. We discuss the problems specific to our implementation in subsequent subsections.

A key challenge is that frameworks such as DPDK have no formal specification. For instance, there is no indication of which operations (start, configure, etc.) are valid in which state (started, stopped, etc.) of a driver. Instead, frameworks tend to evolve organically based on requests from developers.

A challenge specific to using exhaustive symbolic execution is path explosion [3]. While the loop havocing mentioned in §2 is convenient, it can also introduce problems when not done carefully. Havocing replaces all memory that can change during an iteration of the NF's main loop with unconstrained symbolic memory. If havocing makes pointers symbolic, symbolic execution becomes ineffective, because an unconstrained symbolic pointer could point to anywhere in the address space, thus the symbolic execution engine must explore a large number of alternatives. The same problem arises in data structures that use an index into an array, since the index effectively is a pointer.

Another form of path explosion comes from multithreading. In a multithreaded environment in which threads can influence each other's control flow (e.g., via shared memory), every possible interleaving of operations (e.g., the order in which two threads access a shared variable) is a new path.

A less difficult but still time-consuming challenge is modeling. To verify DPDK and the NIC driver, we need models of every layer immediately below them. We therefore have to model the C standard library and the hardware in a detailed way according to the available specifications and manuals. For the NIC we used Intel's official data sheet [15], while for the C standard library we relied on the man pages.

A related challenge is to "future-proof" the models. Our proof of correctness holds for the revisions of the hardware and driver we use. However, if the driver were to write to memory locations that current revisions of the hardware do not use, this could impact the validity of the proof on future hardware. This can happen in two ways: writing to registers that do not exist (since the hardware does not use all of its memory space) or writing to registers that the specification marks as reserved. A future version of the hardware could add registers, or use registers previously marked as reserved. To make sure our proof holds even in future revisions of the hardware, we explicitly model reserved and non-existent registers. Thus, we are confident that the code we proved correct will keep on working correctly on future hardware revisions. It is still possible for future hardware to change in a backwards-incompatible manner, but since this would break existing code, it is unlikely that a hardware manufacturer would make incompatible changes.

3.2 DPDK

The DPDK framework poses specific challenges to symbolic execution. We mitigated some of them by simplifying or reconfiguring DPDK, and addressed the rest by suitably augmenting the symbolic execution engine.

An example of the problems caused by loop havocing is in DPDK's ring buffers. DPDK uses ring buffers to implement memory pools by default. The ring holds an array of items and indices to the start and end of the ring in the array. Adding an element to the ring increments the end index, while removing an element increments the start index. Adding then removing an element causes a change in the ring's internals: both indices change. When this occurs in a loop, havocing the indices during symbolic execution causes path explosion. To address this, we replace the rings with one-item structures: the driver takes the item when receiving a packet, and puts it back after having sent the modified packet. This avoids changing memory during an iteration, thus solving the havoc-induced problem.

Another challenge is DPDK's advanced use of low-level features, such as inline assembly and single instruction, multiple data (SIMD) instructions. The KLEE symbolic execution engine does not support these. We configure DPDK to avoid the use of assembly (theoretically at the expense of performance, but as we show later this is not the case). We add support for basic SIMD instructions to KLEE, such as moving a vector from one memory location to another. We replace the remaining uses by equivalent simpler code.

Finally, DPDK provides utilities for multithreaded NFs such as locks, timers, and thread-local storage. Since our toolchain cannot deal with multithreading, we implement our NF in a single thread and do not use DPDK's thread-related features.

3.3 The NIC Driver

DPDK comes with multiple drivers; we verify the one for Intel's 82599ES, as it is the NIC used for benchmarking in [29]. Its driver, named *ixgbe*, is a generic driver for multiple Intel NICs, with some functions customized for specific models.

The existing tools were not enough to model hardware. The KLEE symbolic execution engine views memory (and thus memory-mapped devices) as an array of bytes with standard read/write semantics. However, while NIC registers are multi-byte words, they hold individual flags and counters of less than a byte each. Furthermore, reads and writes may have side-effects. For instance, the hardware automatically clears some bits after a read operation. Thus, we cannot model registers as simple sequences of bytes in memory. We must emulate hardware behavior precisely, and detect bugs such as a driver trying to write to a read-only hardware bit. To support detailed modeling of registers, we extend KLEE. We add intrinsic functions that replace memory accesses to hardware memory by function calls. This allows us to model bit-level read/write permissions and to execute code on specific reads or writes.

Most registers are storage cells, but some are low-level interfaces to NIC internals. For instance, the 82599 NIC holds an enhanced small form-factor pluggable (SFP+) module accessed through an Inter-Integrated Circuit (I^2C) bus. Thus, the NIC has a register with "data in", "data out", "clock in", and "clock out" bits corresponding to the physical data/clock wires on the bus. We thus model an SFP+ module acting as an I^2C slave as part of the NIC model.

3.4 The C Library

During verification of the NAT in Zaostrovnykh et al. [29], symbolic execution accesses the real, i.e., not modeled, environment. This is problematic because the verification implicitly depends on the specific environment behavior from the machine that ran the verification.

To strengthen our verification claim, we disable calls to the environment during symbolic execution. During verification, we link our NF to the KLEE-uClibc library, which is a small C library customized for verification purposes. We configure the library to disable all environment-related features such as I/O. The remaining functions, such as string comparisons, are pure.

Our models need to cover a wide range of calls, but most of the models are simple. This is because DPDK initializes all subsystems, such as interrupts or timers, even if the NF will not use them. Thus, we can model many operations as no-ops, because they do nothing from the NF's point of view

4 Bugs

One of our hypotheses was that unverified modules of the stack may cause NFs to fail to satisfy their specification. We did find bugs while verifying modules, which we present here as confirmation of our hypothesis. We discuss bugs in DPDK's main library in §4.1, and in its *ixgbe* driver in §4.2.

We found these bugs when we ran the verification after having written our models as described in §3. We reported the bugs to the DPDK maintainers, and wrote simple patches to work around them for our proof.

Replacing code and hardware by models allows us to simulate error cases, no matter how rare, and perform checks for undefined behavior. Thus, we can find failures that happen only rarely in a real environment, and thus are either poorly handled or not handled at all. Such bugs are otherwise hard to reproduce, either because they require complex combinations of circumstances or are outright non-deterministic (such as in code that uses randomness), which delays their fixing.

4.1 Bugs in DPDK

We found one potential crash in DPDK, which we describe below. We also found two cases of undefined behavior, related to memory-mapped files [6] and non-uniform memory access (NUMA) [8].

At initialization time, DPDK may crash if the first 128 CPU cores are all disabled [7]. It looks for the first enabled core, but does not check for the case where it does not find any enabled core before giving up, with a default limit of 128. If no core is available, DPDK performs an out-of-bounds memory access, which is likely to cause a crash. While this scenario is unlikely, it would be hard to understand and debug such behavior, because an environment triggering this bug would likely be perfectly functional in other aspects, and the list of disabled CPU cores is not commonly included in a bug report.

4.2 Bugs in the ixgbe driver

We found five instances of problematic behavior in the *ixgbe* driver, which we describe below. This is surprising given that the code path we explored is straightforward: the NF receives one packet and sends one packet.

We also found issues that are almost certainly omissions from the data sheet. The driver uses register addresses that the data sheet does not document, but the registers are fundamental to receiving packets; since the driver works with a real NIC, those addresses must exist on the NIC.

The most severe bug we found is an incorrect order of operations [13]. The 82599 chip's data sheet states that the driver must configure transmit descriptors before enabling transmission. The *ixgbe* driver does the exact opposite: it first enables transmission, then performs careful checks to make sure transmission is properly enabled, then configures transmit descriptors. This could confuse the NIC hardware in undefined ways, breaking the NAT's specification by arbitrarily dropping or modifying packets.

Undefined behavior may result from copy-paste errors during development. For instance, the *ixgbe* driver uses the name *SW_MNG_SM* for bit 10 of the software-firmware synchronization register [12]. This bit exists, under that name, in other Intel hardware such as the I350 [16], but not in the 82599, thus the driver should not use it. The NIC uses that register for synchronization. If the driver sets a bit that the firmware does not expect, concurrency issues could arise.

Other problems we found are writing to a register at the wrong time [9], incorrectly writing 0 to reserved bits [10], incorrectly writing 1 to a reserved bit [11], and not checking that a register is valid before using it [14].

5 Evaluation

We discuss three metrics: how fast verification runs in §5.1, how much code we verify in §5.2, and the impact of verification on run-time performance in §5.3.

5.1 Verification Time

The verification time for the NF alone on an AMD Opteron 6172 with 48 logical cores at 2.1 GHz is about 5 minutes. This time includes the symbolic execution of the NF code and the validation of the generated files by the Vigor validator.

After expanding the scope of verification to include DPDK and the driver, the time rises to about 75 minutes, a 15× increase. This increase is mainly due to the NIC hardware models: when the driver accesses a register, the symbolic execution engine intercepts the access and calls a function instead. Thus, an otherwise cheap operation becomes expensive. However, we believe this time is reasonable: verification can be run once per release cycle and results in strong guarantees of correctness.

5.2 Verification Coverage

To quantify the additional work done by including DPDK and the NIC driver in the scope of verification, we report in Figure 3 the number of source lines of code (LOC) in each module, as well as its verification coverage. We express the verification coverage as a fraction of LLVM [20] instructions that KLEE executes symbolically, or as 100% for the theorem-proved data structures.

Even though all possible execution paths in the NAT are verified, line coverage is not 100%. The reason for this is that some functions only return an error if their parameters are invalid, and our toolchain proves that the parameters we provide are always valid. Thus, the assertions that check for error values are never triggered, and their associated *exit()* is not reachable.

Most of the code in DPDK and the NIC driver is not reachable from the NF, and thus does not merit verification – as described in §3, this is one of the reasons why our approach can scale. Thus, not surprisingly, our verification covers less than 100% of DPDK and the NIC driver. However, there may be some code in DPDK or the driver that could be reached with more complete models, but those models may cause symbolic execution to not terminate in reasonable time.

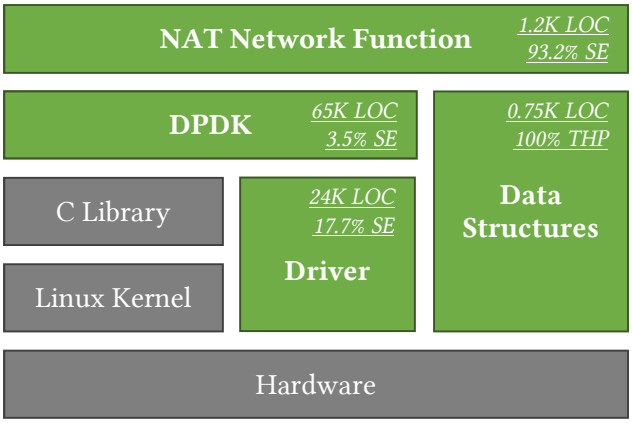

Figure 3: Software lines of code (LOC) and fraction of code that is symbolically executed (SE) or proven using a theorem prover (THP) per module in the NAT stack.

5.3 Performance

To measure performance of the "fully verified" NF, we follow the same methodology as the original benchmarks [29]: measure latency and throughput with a varying number of flows but a constant rate of packets.

Our results are simple: performance does not change relative to the NF with an unpatched DPDK [29]. Benchmarking our modified code yields results that are well within the jitter of the original measurements.

Since our bugfixes do not fundamentally alter DPDK, this is not particularly surprising. We replace some high-performance code with simpler, slower code, but this has a negligible effect on the simple NAT. Performance might change for NFs that, for instance, use concurrent packet handling or batching.

6 Discussion

Our extended proof of NF correctness gives NF developers more confidence in the correctness of their code, even in the presence of timing issues or unusual environments. It also gives developers more confidence when using a new framework such as DPDK, since the proof verifies the framework as needed by the NF.

We believe that our method is reusable in other contexts. Our main observation – that we can symbolically execute DPDK and NIC drivers – can be applied to other kernel-bypass or equivalent frameworks.

However, our proof does not cover the entire stack, nor does it cover all cases in the modules it applies to. We state our assumptions in §6.1, explain the applicability of our work to real-world NFs in §6.2, discuss limitations of our approach in §6.3, and propose solutions to mitigate these limitations in §6.4.

6.1 Assumptions

Our proof makes assumptions about the NF, DPDK, and the software and hardware environment.

We assume the NF to be implemented as prescribed in [29]. Namely, it needs to adopt a stateless/stateful separation, use the libVig library of verified data structures, and limit its use of certain dangerous C operations such as pointer arithmetic. All of these assumptions can be mechanically checked.

We assume that the few DPDK methods which we replace for verification purposes are correct. These are low-level methods, such as a "memory copy" method that uses SIMD instructions to copy large blocks of memory at a time. They are so commonly used in DPDK that we believe any bugs would have been noticed.

Our environment model includes errors and unusual values, but it is not exhaustive in that respect. We do not return symbolic data for all environment properties. For instance, if the memory page size returned by *getpagesize()* was symbolic, memory-related operations would have too many paths for symbolic execution to finish in reasonable time, so we assume that the page size is always 4 KB.

Our NIC hardware model is not exhaustive. If the hardware malfunctions, reads could produce arbitrary data, which we do not model. Normal NIC operation causes the driver to read registers hundreds of times. If each read could return any value, the number of resulting paths would cripple our toolchain – even if we limited the symbolic execution engine's running time, it would be unlikely to explore interesting paths such as correct hardware initialization. To avoid this problem, we assume the hardware behaves correctly.

We believe these assumptions are reasonable even though they weaken our claim. Because of the path explosion problem, our models cannot be as precise as we would like them to be.

We believe our assumptions to be reasonable for virtually all environments that would want to run a verified NF, even though they do weaken the proof.

The assumption that is most likely to break is the one concerning the hardware, since NICs are complex pieces of hardware, with lots of features. Nevertheless, informal discussion with network operators indicate that software reliability is much further up their list of concerns than hardware reliability.

6.2 Applicability to Real-World NFs

There are three main issues when applying our work to existing real-world NFs: data structure complexity, unsupported low-level code, and multithreading.

Existing NFs use their own data structures, which are not verified. Our approach requires that the NFs use verified data structures provided by libVig. This library contains a map, a port allocator and a flow lifetime manager, which are enough for many basic NFs (e.g., NATs and firewalls), but not all (e.g., an IP router requires a longest-prefix-match table). However, this is not a fundamental limitation: writing new verified data structures for libVig can be done with some human effort.

Kernel-bypass frameworks and tools such as Snort [26], netmap [25] and SoftNIC [27] use low-level code and inline assembly for performance-sensitive operations (e.g., computing hashes or reading hardware timestamps). Our toolchain does not support symbolically executing such low-level code. This is not a fundamental limitation either, although the breadth of possible low-level features may require significant engineering effort to support.

Finally, many modern NFs are multithreaded. This is a more fundamental limitation of our approach because symbolic execution is hard to apply directly to multithreaded code. We plan to work on this aspect in the future.

6.3 Limitations

There are three remaining unverified modules in the NF stack, and the Vigor toolchain is not verified either.

We do not verify the C library, the Linux kernel, and the hardware. A failure in any of those may compromise either the safety or the liveness of the NF; that is, cause the NF to break its semantic properties (e.g. send an incorrect packet), or to stop working (e.g. enter an infinite loop), respectively.

We can protect the NF against some issues, but not all. Since the environment model returns symbolic data, we can prove that the NF will behave correctly even if its software environment returns nonsensical values. However, if the Linux kernel were to take over the NIC and instruct it to send an incorrect packet (e.g. due to a bug in the kernel or an exploited security vulnerability), the NF code would not even see the problem, let alone prevent it. Similarly, we cannot prove that the Linux scheduler will not put the NF to sleep and never wake it up, or that the hardware will not halt and catch fire.

Proving Linux kernel correctness is currently infeasible, as its contributors did not write its millions of lines of code with ease of verification in mind.

Proving hardware correctness may be possible, but would need a different toolchain from ours. Even if we formally verified the hardware, problems such as random bit flips from cosmic rays would still exist.

We do not verify the Vigor toolchain. Neither KLEE, nor VeriFast, nor the validator are verified. *Clang* [20], the C compiler we use both for verification (by translating C code to a format KLEE understands) and execution (by translating C code to executable machine code), is not verified either.

In summary, our trusted code base (TCB) is made up of the C library, the Linux kernel, the hardware, the Vigor proof toolchain, and the C compiler.

6.4 Proposed Solutions

We could verify the C library in the same way we verify DPDK, by modeling the Linux kernel and including the C library in the proof.

Then, the NF could be run on an OS kernel stripped down to the minimal subset needed to start up DPDK. This would perhaps enable us to symbolically execute the OS kernel as well. Such an approach would mimic what we did for DPDK: only prove correctness for code that the NF actually uses. This stripped-down OS kernel could use the environment models we wrote as actual implementations, such as having a single user instead of supporting multiple users.

An alternative would be to adapt DPDK to use an already verified special-purpose kernel such as seL4 [18]. Since DPDK bypasses the kernel already, it is not too dependent on Linux. However, the result would be a merge of two different approaches to verification, which might cause problems when stitching the two together.

7 Related Work

SLAM [1] analyzes Windows drivers based on specific rules, with such a low false positive rate that it is mandatory for driver certification. Post and Küchlin [21] adapted it to Linux drivers. These projects have a wider applicability than ours, but check specific rules for code correctness, such as proper lock usage; they do not prove that a driver satisfies a specification.

DDT [19] and SymDrive [23] test drivers using symbolic execution with symbolic hardware models. A developer can use those tools to check for basic correctness properties such as crash-freedom, but not semantic properties. Their models are more thorough than ours: they implement fully symbolic hardware, including a limited version of symbolic interrupts. This does not allow them to prove the drivers correct, but it is efficient at finding bugs.

8 Conclusion

We extend the correctness proof of an existing formally verified NAT to include its kernel-bypass framework it uses, and the NIC driver. We used the DPDK framework and its driver for the Intel 82599ES NIC, but we believe that our approach can generalize to other frameworks and drivers.

Our contribution is a formal proof of correctness for DPDK and a driver in the context of a given NF. We do not make claims about other drivers, but we guarantee the driver we verified to be correct in the context of the NAT we verified.

Compared to the original proof, we offer a stronger guarantee of correctness as we move code out of the trusted base. We confirm the usefulness of our proof by finding bugs in both DPDK and the NIC driver.

Acknowledgements

We thank Katerina Argyraki, Rishabh Iyer, Viktor Kunčak, Luis Pedrosa, Betty Pirelli, and the anonymous reviewers for their helpful comments and suggestions.

References

[1] Ball, T., Bounimova, E., Kumar, R. and Levin, V. 2010. SLAM2: Static driver verification with under 4% false alarms. *Formal Methods in Computer Aided Design (FMCAD '10)* (Austin, TX, 2010), 35–42.

[2] Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B. and Leino, K.R.M. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. *International Symposium on Formal Methods for Components and Objects (FMCO '05)* (Nov. 2005), 364–387.

[3] Boonstoppel, P., Cadar, C. and Engler, D. 2008. RWset: Attacking Path Explosion in Constraint-Based Test Generation. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)* (Berlin, Heidelberg, 2008), 351–366.

[4] Cadar, C., Dunbar, D. and Engler, D.R. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)* (Berkeley, CA, USA, 2008), 209–224.

[5] DPDK: Data Plane Development Kit: 2018. *http://dpdk.org/*. Accessed: 2018-02-12.

[6] DPDK bug 18: mmap with MAP_ANONYMOUS should have fd == -1: *https://dpdk.org/tracker/show_bug.cgi?id=18*. Accessed: 2018-04-01.

[7] DPDK bug 19: Crash on initialization if first RTE_MAX_LCORE cores are disabled: *https://dpdk.org/tracker/show_bug.cgi?id=19*. Accessed: 2018-04-01.

[8] DPDK bug 20: Undefined behavior caused by NUMA function in eal_memory: *https://dpdk.org/tracker/show_bug.cgi?id=20*. Accessed: 2018-04-01.

[9] DPDK bug 21: Ixgbe driver changes FCTRL without first disabling RXCTRL.RXEN: *https://dpdk.org/tracker/show_bug.cgi?id=21*. Accessed: 2018-04-01.

[10] DPDK bug 22: Ixgbe driver sets RDRXCTL with the wrong RSCACKC and FCOE_WRFIX values: *https://dpdk.org/tracker/show_bug.cgi?id=22*. Accessed: 2018-04-01.

[11] DPDK bug 23: Ixgbe driver writes to reserved bit in the EIMC register: *https://dpdk.org/tracker/show_bug.cgi?id=23*. Accessed: 2018-04-01.

[12] DPDK bug 24: Ixgbe driver sets unknown bit of the 82599's SW_FW_SYNC register: *https://dpdk.org/tracker/show_bug.cgi?id=24*. Accessed: 2018-04-01.

[13] DPDK bug 25: Ixgbe driver sets TDH register after TXDCTL.ENABLE is set: *https://dpdk.org/tracker/show_bug.cgi?id=25*. Accessed: 2018-04-01.

[14] DPDK bug 26: Ixgbe driver does not ensure FWSM firmware mode is valid before using it: *https://dpdk.org/tracker/show_bug.cgi?id=26*. Accessed: 2018-04-01.

[15] Intel® 82599 10 GbE Controller Datasheet: 2016. *https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf*. Accessed: 2018-02-12.

[16] Intel® Ethernet Controller I350 Datasheet: 2017. *https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/ethernet-controller-i350-datasheet.pdf*. Accessed: 2018-02-12.

[17] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W. and Piessens, F. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. *Third International Conference on NASA Formal Methods* (Berlin, Heidelberg, 2011), 41–55.

[18] Klein, G., Norrish, M., Sewell, T., Tuch, H., Winwood, S., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K. and Kolanski, R. 2009. seL4: Formal Verification of an OS Kernel. *22nd ACM symposium on operating systems principles (SOSP '09)* (New York, NY, USA, 2009), 207.

[19] Kuznetsov, V., Chipounov, V. and Candea, G. 2010. Testing Closed-Source Binary Device Drivers with DDT. *2010 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), 12.

[20] Lattner, C. 2008. LLVM and Clang: Next Generation Compiler Technology LLVM: Low Level Virtual Machine. *The BSD Conference* (2008).

[21] Post, H. and Küchlin, W. 2007. Integrated Static Analysis for Linux Device Driver Verification. *Integrated Formal Methods* (Berlin, Heidelberg, 2007), 518–537.

[22] Release Notes - Data Plane Development Kit: 2018. *https://dpdk.org/doc/guides/rel_notes/*. Accessed: 2018-02-12.

[23] Renzelmann, M.J., Kadav, A. and Swift, M.M. 2012. SymDrive: Testing Drivers without Devices. *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)* (Hollywood, CA, 2012), 279–292.

[24] RFC 3022: Traditional IP Network Address Translator (Traditional NAT): 2001. *https://www.ietf.org/rfc/rfc3022.txt*. Accessed: 2018-02-12.

[25] Rizzo, L. 2012. NetMap: A novel framework for fast packet I/ O. *21st USENIX Security Symposium (USENIX Security '12)* (2012), 101--112.

[26] Roesch, M. 1999. Snort: Lightweight Intrusion Detection for Networks. *13th Systems Administration Conference (LISA '99)* (1999), 229–238.

[27] SoftNIC: A Software NIC to Augment Hardware: 2015. *http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html*.

[28] VigNAT home repository - GitHub: *https://github.com/vignat/vignat*.

[29] Zaostrovnykh, A., Pirelli, S., Pedrosa, L., Argyraki, K. and Candea, G. 2017. A Formally Verified NAT. *Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)* (New York, New York, USA, 2017), 141–154.