HOL4P4.EXE: A Formally Verified P4 Software Switch

Didrik Lundberg[1,2] [0000−0001−9921−3257] and Roberto Guanciale[1,3] [0000−0002−8069−6495]

[1] KTH Royal Institute of Technology, Lindstedtsvägen 5, 100 44 Stockholm, Sweden
{didrikl, robertog}@kth.se
[2] Saab AB, Nettovägen 6, 175 41 Järfälla, Sweden
[3] Digital Futures, Osquars Backe 5, 100 44 Stockholm, Sweden

Abstract. The emergence of programmable network elements and their usage in critical infrastructure is increasing the demand for formal guarantees of their correctness. This paper presents the first P4 software switch connected by proof to a formal semantics. This is accomplished by adjusting and optimizing the HOL4P4 semantics to create an efficient interpreter that can be compiled using the verified CakeML compiler. We demonstrate practical performance in a set of experiments, achieving several orders of magnitude improvement in throughput over existing semantics-based interpreters, with only small performance penalties compared to the BMv2 reference switch written in C++.

Keywords: P4, Theorem Proving, Formal Verification

1 Introduction

Software-defined networking gives hardware owners the power to fine-tune the data plane processing in a protocol-independent manner. The typical use case for leveraging this flexibility is to gain a rapid development cycle (as opposed to that of non-programmable hardware) and to support specialized infrastructure with custom performance and security solutions. The P4 language is the de facto standard for programmable network elements, and P4 compilers target a variety of platforms from terabit-bandwidth switches [8] to network interface controller (NIC) cards. In addition, P4 programs can run on commodity CPUs by various means [21].

When executing a high-level language like P4, the compilers, interpreters, and runtimes involved may deviate from the language's intended semantics or introduce bugs, especially if they are complex, written in an unsafe language, and heavily performance-optimized. The goal of this work is to address these problems by developing a formally verified P4 software switch, hereafter referred to as HOL4P4.EXE[4]. A *verified software switch* is a provably correct implementation of P4 programs and architecture models according to a high-level formal semantics.

[4] The source code is available in the HOL4P4 Github repository https://github.com/kth-step/HOL4P4 at the tag `VSTTE2025`.

Our main strategy is to compile the HOL4P4 semantics, a formal semantics of the P4 language developed in the HOL4 interactive theorem prover, into an executable interpreter using the CakeML verified compiler. This enables us to bridge the gap between formal verification and practical deployment of P4 software switches. This paper makes four key contributions:

First, we modify the original HOL4P4 semantics to be compatible with CakeML extraction and update the symbolic executor to enable end-to-end proofs of program behavior at the machine code level.

Second, we profile the resulting interpreter and identify several performance bottlenecks in the CakeML-generated code. Guided by these insights, we develop an optimized version that significantly reduces extraction time and yields a more efficient binary. This offers practical guidance on building performant interpreters via executable semantics.

Third, we extend the verified interpreter with a raw-socket FFI, resulting in a usable software switch. We further integrate it with Mininet [10], demonstrating that HOL4P4.EXE can operate within realistic testing environments.

Finally, We evaluate HOL4P4.EXE side-by-side with existing solutions, showing that it outperforms current semantics-based implementations and scales well with increasing packet sizes.

HOL4P4.EXE is the first P4 software switch with formal verification guarantees. Our results demonstrate that verified systems can achieve practical performance, advancing the goal of high-assurance networking.

2 Related Work

We use HOL4P4 [1] as formal semantics of P4. HOL4P4 is a heapless semantics written in the interactive theorem prover (ITP) HOL4 and has an executable formulation, that is, a version that can be evaluated in the ITP, computing the result of running a P4 program. To facilitate verification of P4 programs, HOL4P4 has been extended with a symbolic executor [14]. In this work we also use CakeML [9], a verified compiler implemented in HOL4 which compiles the CakeML language, a dialect of Standard ML. CakeML is also capable of verified extraction of CakeML code from HOL4 definitions.

There are a number of software switches that enable running a P4 program on a regular CPU. BMv2 [18] is a "reference P4 software switch implementation". However, unlike HOL4P4.EXE, BMv2 is not based on a formal semantics. Petr4 [4] is a P4 semantics and formalization in OCaml. There exists a version of petr4 which can also be compiled and used as a software switch, e.g. with Mininet [6], making it directly comparable to HOL4P4.EXE. The petr4 semantics has also been ported to the Coq ITP [4,19,26], however there is no formal connection between the petr4 software switch and the ITP semantics.

The Coq ecosystem also includes the verified compiler CompCert [10], and a fully verified C code extraction mechanism from Coq, which does not currently exist, would offer a similar pathway to a verified switch as the one presented in this paper.

PfComp [3] is another work that applies both theorem-proving and verified compilation to networking. It uses Coq to construct a verified compiler from firewall policies to Clight, which can in turn be compiled using CompCert. In contrast to the P4 programs used to program HOL4P4.EXE, the policy language of PfComp does not include packet parsing, modification and deparsing, nor outcomes beyond acceptance or rejection.

In addition to the ITP-based verification tools [5,19,26,27,14], multiple non-proof-producing tools have been made to verify P4 programs [23,17,11]. Verification tools are complementary to a verified switch implementation, whose main purpose is to transfer verified properties from source code to machine code.

3 System Description

The core of HOL4P4.EXE is a verified P4 interpreter, obtained by adapting the existing HOL4P4 semantics and architectural models (V1Model and eBPF), and then compiling the semantics function via the verified CakeML compiler. The main challenge lies in refining the semantics to support CakeML extraction while preserving the soundness of the verification tools, in identifying bottlenecks and optimizations that can guide the development of efficient executable semantics, and integrating the interpreter in a way that minimizes the trusted computing base (TCB). The version of the semantics with the optimizations described in Section 3.1 is called the *optimized semantics* in the rest of the paper. We have also implemented in HOL4 a translator from the abstract syntax tree (AST) of standard HOL4P4 programs and initial states to those of the optimized semantics.

Several aspects of the HOL4P4 semantics make it a good candidate as basis for our executable interpreter. In particular, the semantics models the specific P4 calling convention, which forbids cross-function variable references, via a heapless design: as opposed to allocating local variables in a global heap, HOL4P4 keeps them in function frames that are disposed of upon function return.

Figure 1 shows an overview of HOL4P4.EXE. The verified interpreter is used together with a wrapper program in CakeML and a foreign function interface (FFI) library to communicate over Linux raw sockets to obtain a usable software switch. This paper's code contributions are shown in green: the architecture models and semantics are adapted from prior work. The parts below the dashed line exist inside the theorem prover HOL4, while the parts above it are outside.

Note that the P4 program and the table configurations are statically embedded in the final HOL4P4.EXE executable. This is achieved by extracting the HOL4 term representing the AST of the program to CakeML, which is then compiled into machine code that reconstructs the same AST at start time. As a result, the interpreter begins execution with the target P4 program already parsed and ready for packet processing.

From a usability perspective, standard P4 programs can be translated into HOL4P4 programs by using the .p4 parser functionality of petr4[4] and feeding the JSON result to the HOL4P4 import tool. Also, while this paper only describes

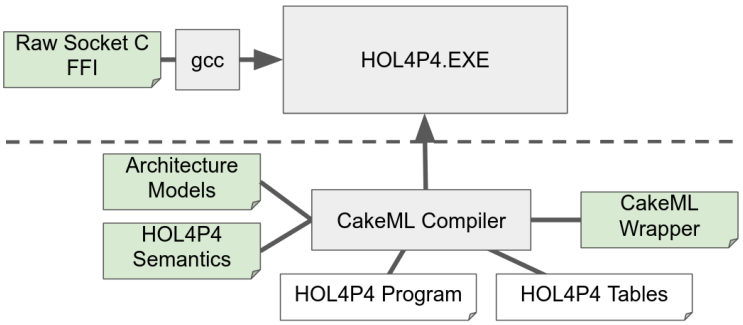


Fig. 1: HOL4P4.EXE system overview

compilation to x86, the HOL4P4.EXE toolchain supports compilation to the RISC-V and ARMv8 ISAs.

3.1 Refining the Semantics for CakeML

In order to make extraction to CakeML from HOL4 possible, all instances of Hilbert choice were removed from the semantics. The Hilbert choice was initially used for indefinite values of uninitialised variables. The resulting semantics zeroes all new variables, which is sound with respect to the P4 specification and indeed guaranteed by some P4 platforms. This stricter initialisation scheme preserves security properties proved by the symbolic executor, but when used as a simulator, the software switch may have more restricted behaviour than generally allowed. Moreover, the partial function definitions allowed by HOL4 were replaced by option-type functions or guarded behind checks. Partial functions may also be extractable by CakeML, but require manual proofs showing that they always yield results where they are used. A quirk of P4 is the heavy usage of bitstrings of non-standard widths (e.g., 3 or 11) and operations on these. To be able to extract the semantics for arithmetic, this was rewritten to using just bitstrings and numbers instead of fixed-width word types. The rarity of numerical computations in P4 programs means that trading off performance for generality here is not significant for overall performance in practice.

Optimized semantics For increasing performance, all string identifiers in the semantics and architecture models (variable names, function names, field names, parser state names, table names and string constants) were replaced by native-size words, depending on compilation target: typically 64 bits, with 32 and 16 also supported. The complete evaluation is in Section 4, note that the comparisons here use the same test setup.

This led to around 60% increased throughput and 30% lowered latency, fairly consistent over different programs. Most extraction times stayed constant, with only a small 20% decrease in extraction time for the AST of the largest program. In our experiments, the compilation step of HOL4P4.EXE always stayed under

20s for all tested programs except for the string version of the largest program in Section 4, which took 5m. The heapless design of the HOL4P4 semantics means that many small variable maps are used instead of one large one. Here, the asymptotic benefits of efficient implementations like red-black maps become less important compared to the association lists used by HOL4P4. However, the relative speedup gained by switching identifiers from strings to native-length words is greater when using association lists, since they require more identifier comparisons per lookup.

Another important optimization was the on-demand conversion of byte arrays to the bit representation used by HOL4P4. Since the raw socket FFI yields incoming packets as byte lists, and since only the headers (and not the data payload) is supposed to affect the program, it is more efficient and scalable to only convert the necessary bytes to bits when extracting headers and vice versa when emitting. This led to 4x the throughput of the unoptimized version for 1518-byte packets, with almost indistinguishable throughput for 64-byte packets.

The most performance-critical operation in P4 programs is table matching, where, e.g., IP addresses are matched against table entries, which may include ranges and bit masks. The HOL4P4 semantics performs these operations using the same bitstring-based definitions written for the arithmetic semantics: writing a separate implementation using 64-bit word comparisons reduced performance by half when matching against tables with 1000 range-type entries. This suggests that at least on x86-64, the HOL4P4-style bitstring comparisons are more efficient than comparisons from HOL4's theory of machine words when compiled by CakeML.

3.2 Formal Guarantees

The guarantees of CakeML apply to two stages: CakeML code extraction from HOL4 definitions and binary compilation from CakeML code.

The HOL4 definitions extracted to CakeML consist of the entire semantics, the architecture models as well as the AST of the P4 program to run: in the wrapper program, only the top-level semantics function `cake_exec` is used (on the extracted *prog* and *state*, with incoming packets added). Theorem 1 states the correctness of the extraction relative to a correspondence between CakeML and HOL4 values [15]. This step is shown with red arrows in Figure 2.

Theorem 1. *When given an application of the CakeML HOL4P4 semantics to arguments corresponding to HOL4 terms, the CakeML operational semantics will terminate with a value that corresponds to the application of the HOL4 HOL4P4 semantics to those terms.*

The compilation guarantees also preserve the semantics of the CakeML wrapper code, as stated in Theorem 2 [9,24], a simplified and specialised version of the top-level correctness theorem of CakeML. This property relies on a few assumptions: notably, that any FFI functions written externally obey CakeML's requirements, specifically that they only write to their proper memory regions

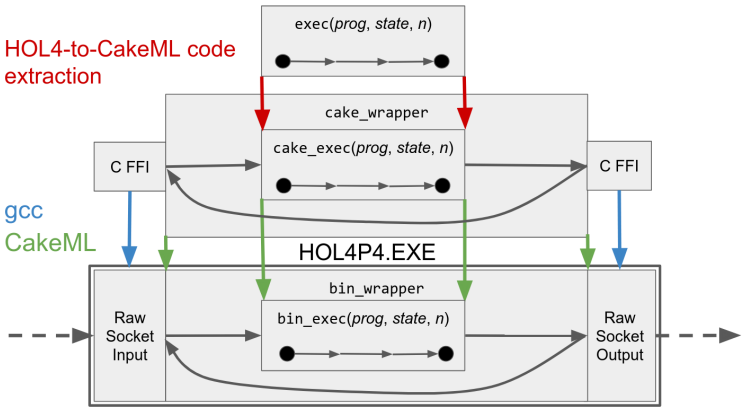


Fig. 2: HOL4P4.EXE compilation overview

when running the binary. Regarding memory, note that CakeML includes garbage collection that is formally verified to uphold the compiler correctness theorem.

Theorem 2 (Top-level compiler correctness). *The HOL4P4.EXE binary produced by a successful evaluation of the CakeML compiler function will either*

- *behave exactly according to the observable behaviour of the CakeML source code according to its semantics, or*
- *behave the same as the CakeML source code up to some point at which it terminates with an out-of-memory error.*

Component	Written in	Size (LoC)
FFI	C	551
Wrapper	CakeML	521
HOL4P4 executable semantics	HOL4	1650
Architecture models	HOL4	1331
Auxiliary definitions	HOL4	808
		3769

Table 1: Sizes of HOL4P4.EXE components

The fact that we not only enabled CakeML extraction of the executable semantics but also updated the metatheory and symbolic executor of Lundberg et al. [14] to accommodate these changes, means we can use the symbolic executor on the exact same semantics and programs that then are deployed in the verified P4 software switch. Since the verified compiler preserves functional properties, this specifically means that it preserves properties proved using the symbolic executor. Accordingly, an *end-to-end proof* is obtained in the sense that the

binary fragment running the P4 program also obeys the high-level properties you have proved using the symbolic executor.

3.3 Implementation and TCB

Information about the various components is shown in Table 1. Code sizes are calculated using `cloc` and provide an indication of the size of the TCB. Components in the TCB require different degrees of trust; the FFI and the system calls have to be wholly trusted, whereas the CakeML compiler ensures the wrapper behaves according to the CakeML semantics, assuming the FFI is well-behaved. The correspondence in Theorem 1 could also be considered part of the TCB.

In theory, the FFI could get stuck in an infinite loop, or completely upend the other guarantees by writing to arbitrary memory locations. However, the raw socket FFI code size is very small, and the dependencies consist of compile-time constants and structure definitions in addition to system calls, contributing almost no additional executable code in the final binary. In addition to the listed components, standard library functions of the respective languages that are used, Linux, and its device drivers should also be considered part of the TCB.

Notably, if a P4 program is proven to satisfy a property with respect to the HOL4P4 semantics, verified compilation ensures that this property is preserved in the final binary. From the perspective of this property, the semantics and architecture model are no longer part of the TCB, as the compilation correctness theorem guarantees that the binary faithfully implements the semantics, which has been proved to guarantee the property.

The contributions in this paper also consist of changes to the existing HOL4P4 theories and tools: approximately 20000 lines of code (LoC) were added in total, according to a Git diff analysis. Around 25% of these are changes to existing theories and libraries.

For comparison, the petr4 OCaml model can be estimated to be about 12000 LoC[5] and the BMv2 model about 16500 LoC[6]. Note that these rely on the OCaml and C++ compilers, respectively, as well as any external libraries used.

4 Performance Comparisons

The main goal of this section is to isolate aspects of operation in order to obtain performance bounds and identify or rule out potential bottlenecks. Unless noted otherwise, all HOL4P4.EXE results refer to the version using the optimized semantics.

To evaluate our system experimentally, we have selected the following P4 programs, all using the V1Model architecture:

[5] Estimated using `cloc *.ml` on the `lib` directory of the latest release (0.1.3).

[6] Estimated using `cloc *.c *.cpp *.h` on the `src/bm_sim` directory of the latest release (0.15.0).

1. `port_swap.p4`: This small program extracts and emits Ethernet and IPv4 headers unchanged, and looks at the ingress port: if it is 1, the egress port is set to 2, otherwise to 1. This is meant to illustrate the baseline cost of P4 interpretation and the overhead of the basic V1Model pipeline implementation.
2. `vss-example.p4`: A medium-sized P4 program from the "Very Simple Switch" (VSS) example of the P4 Specification [25] which performs basic switch functionality. For these tests, it has been adapted to the V1Model architecture.
3. `fabric_border_router.p4`: A larger (2816 LoC) real-world industry program used at Google [22]. Note that HOL4P4.EXE only has placeholder models for timing-dependent externs such as meters whose real-world implementations rely on system clocks.

The sizes of the programs cover a range of P4 program sizes from minimal to real-world industry applications, and so are suitable to determine scalability. To provide context for these measurements, the minimum bandwidth listed under Zoom system requirements for a 720p HD video call is 1.2Mbps, while the minimum for a voice call is 80kbps [29]. Netflix recommends 5Mbps or higher for 1080p video streaming [16].

All measurements were performed on a laptop with an Intel® i7-8550U CPU. This involved first setting up small virtual networks in a Mininet-like fashion and configuring the environment to maximize the accuracy of the measurements, after which Pktgen-DPDK [28] was used to run test suites scripted in Lua.[7]

The closest comparable (but unverified) tool to HOL4P4.EXE is petr4[8], the only other software switch based on a formal semantics known to the authors, written in OCaml. HOL4P4.EXE consistently outperforms petr4, sometimes by several orders of magnitude. While HOL4P4.EXE does not yet match the performance of the reference software switch BMv2, implemented in C++, it shows that the formal guarantees can be achieved without sacrificing real-world usability.

4.1 Extraction and Compilation Measurements

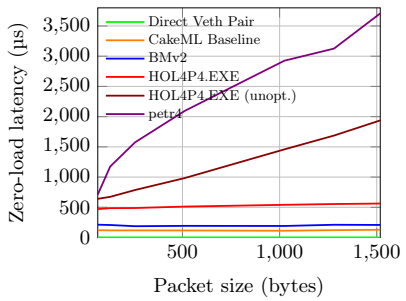
The verified code extraction of the semantics and the V1Model architecture model takes 10m, while extraction of the `vss-example.p4` program takes 4m35s with no additional table entries and 5m with 100 additional entries. Note that the extraction of table entries can be separated from that of the program, so that recompiling the same switch with new table entries can be done in seconds. The code extraction of `port_swap.p4` takes 3m25s and that of `fabric_border_router.p4` 6m. The final step of compilation to binary only takes a few seconds.

[7] The testbed is available in the Github repository https://github.com/kth-step/swswitch-perf at the tag `VSTTE2025`.

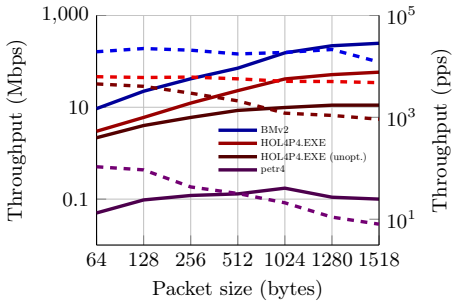
[8] Using the Mininet-capable version in the branch `mininet-integration` that was presented at CAV 2021 [6]. Printing of debug messages and related calculations were commented out in the source code to increase performance.

The BMv2 binary is 2.5 MB and the petr4 binary is 27.5 MB. For `port_swap.p4`, the HOL4P4.EXE binary is 970 kB, for `vss-example.p4` it is 1.1 MB and for `fabric_border_router.p4` 2.0 MB. Notably, since the program is linked in the binary instead of parsed at run-time (as for petr4 and BMv2), the HOL4P4.EXE size increases with the size of the P4 program that is used, but remains smaller than for the alternatives.

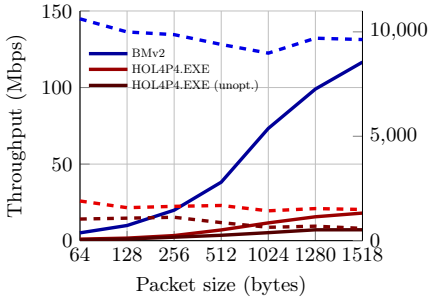
4.2 Zero-load UDP Latency



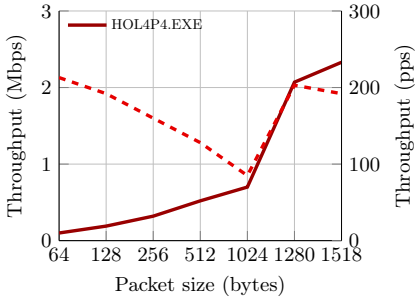
(a) Zero-load latency comparison for `port_swap.p4`



(b) Throughput comparison for `port_swap.p4`. Note the logarithmic scale on the y-axis.



(c) Throughput comparison for `vss-example.p4`



(d) Throughput of HOL4P4.EXE for `fabric_border_router.p4`

Fig. 3: Performance measurements. Solid and dashed lines are scaled according to the left and right y-axes, respectively.

The zero-load latency of different P4-programmable software switch solutions running `port_swap.p4` is shown in Figure 3a. All these communicate over Unix raw sockets using the same test setup. Two other implementations are provided for comparison: a direct veth pair (virtual network cables linking Linux network

namespaces), which has a constant 2 µs latency; and the baseline CakeML program, which simply uses the raw socket FFI to forward a packet like `port_swap.p4` would. The baseline program has a constant latency of around 120 µs, which can be considered a lower bound for the latency that could be achieved by verified compilation using the same CakeML FFI.

Among the software switches, BMv2 achieves a roughly constant latency of 200 µs regardless of packet size, whereas both petr4 and HOL4P4.EXE have a linearly increasing latency, with HOL4P4.EXE almost achieving constant latency. The performance of petr4 seems to slowly degrade the longer the switch is running, implying issues with garbage collection. The cost of the HOL4P4.EXE semantics interpreting `port_swap.p4` is roughly 520 µs, which is obtained by subtracting the latency of the CakeML baseline solution from that of HOL4P4.EXE.

The conclusions from measuring zero-load latency for `vss-example.p4` are similar to those from the `port_swap.p4` case, with BMv2 and HOL4P4.EXE achieving a near-constant 360 and 1400 µs, respectively, and petr4 scaling from around 2400 to 14000 µs. The large `fabric_border_router.p4` program incurs a larger latency of 16.5 ms when using HOL4P4.EXE.

We performed throughput testing using an RFC2544-derived method. The results for `port_swap.p4` are shown in Figure 3b. Due to the large performance differences between our approach and petr4 (multiple orders of magnitude), we used a logarithmic scale to better visualize both datasets in the same graph. From this, we observe that HOL4P4.EXE achieves around 3 Mbps for the 64-byte case and 58 Mbps for the 1518-byte case. In comparison, petr4 achieves no more than 0.15 Mbps in the best case - three orders of magnitude below HOL4P4.EXE.

The throughput results of the VSS example, shown in Figure 3c, are similar. The HOL4P4.EXE maximum throughput is around 18 Mbps, with BMv2's advantage increasing somewhat at 117 Mbps. The throughput of petr4 was too low to accurately measure with the lowest packet generation rates of Pktgen-DPDK, hence its absence from Figure 3c and Figure 3d. BMv2 was not able to run `fabric_border_router.p4`, and the unoptimized version of HOL4P4.EXE had too low throughput.

4.3 Table Usage

In order to benchmark performance over different table sizes, the `vss-example.p4` program was used with different numbers of random, non-matching, non-overlapping entries (32-bit IPv4 addresses) inserted before the matching entries in an existing table. The packet size for this test was 1518 bytes. BMv2 manages to keep latency roughly constant, while each additional table entry contributes around 3 µs for HOL4P4.EXE and 1.6 µs for petr4.

5 Conclusions and Future Work

We have presented HOL4P4.EXE, the first P4 software switch built with formal correctness guarantees, and demonstrated that it outperforms a previous

semantics-based solution by orders of magnitude and fulfils minimum requirements for practical use. In summary, efficient prototype trusted execution can be achieved via verified compilation of executable semantics as long as the optimizations are in place.

A formal proof that the optimized version of the HOL4P4 semantics preserves the original semantics under transformation to the optimized state is currently in progress. Also, we could prove the CakeML wrapper functionally correct by using the existing characteristic formulae-based tools for CakeML [7,2]. Our experimental results in Figure 3a show promise that future CakeML-based verified compilation techniques that do not rely on compiling an interpreter could outperform BMv2. Firstly, the virtual loop-freeness of P4 programs means one could take the result of symbolic execution [14] and use it to define a function in HOL4 with the symbolic input as parameter, which is then extracted to CakeML (the "CakeML baseline" graph in Figure 3a is from an unverified version of this approach). Secondly, one could use the HOL4P4 semantics to write a verified compiler to another language, for which there already exists a verified compiler: ideal candidates for this are the low-level systems language Pancake [20] or Verilog [13,12]. However, this approach might require a larger proof effort.

Acknowledgments. This work was in part financially supported by Digital Futures, and in part by the SEMLA project financed by Vinnova (Sweden's Innovation Agency). We would also like to thank Magnus Myreen for valuable correspondence regarding the usage of CakeML.

References

1. Alshnakat, A., Lundberg, D., Guanciale, R., Dam, M.: HOL4P4: Mechanized small-step semantics for P4. Proceedings of the ACM on Programming Languages **8**(OOPSLA1), 223–249 (2024). https://doi.org/10.1145/3649819
2. Åman Pohjola, J., Rostedt, H., Myreen, M.O.: Characteristic formulae for liveness properties of non-terminating CakeML programs. In: 10th International Conference on Interactive Theorem Proving (ITP 2019). pp. 32–1 (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.32
3. Chavanon, C., Besson, F., Ninet, T.: Pfcomp: A verified compiler for packet filtering leveraging binary decision diagrams. In: Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 89–102. CPP 2024 (2024). https://doi.org/10.1145/3636501.3636954
4. Doenges, R., Arashloo, M.T., Bautista, S., Chang, A., Ni, N., Parkinson, S., Peterson, R., Solko-Breslin, A., Xu, A., Foster, N.: Petr4: formal foundations for P4 data planes. Proc. ACM Program. Lang. **5**(POPL) (2021). https://doi.org/10.1145/3434322
5. Doenges, R., Kappé, T., Sarracino, J., Foster, N., Morrisett, G.: Leapfrog: Certified equivalence for protocol parsers. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 950–965. PLDI 2022 (2022). https://doi.org/10.1145/3519939.3523715
6. Foster, N., Tahmasbi Arashloo, M., Parkinson, S.: CAV'21 P4 verification tutorial (2021), https://github.com/cornell-netlab/cav21-tutorial

7. Guéneau, A., Myreen, M.O., Kumar, R., Norrish, M.: Verified characteristic formulae for CakeML. In: Programming Languages and Systems. pp. 584–610 (2017). https://doi.org/10.1007/978-3-662-54434-1_22
8. Intel Corporation: *P4 16* Intel® Tofino™ native architecture - public version (2021), https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf
9. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. SIGPLAN Not. **49**(1), 179–191 (2014). https://doi.org/10.1145/2578855.2535841
10. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: CompCert - A Formally Verified Optimizing Compiler. In: ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress (2016), https://inria.hal.science/hal-01238879
11. Liu, J., Hallahan, W., Schlesinger, C., Sharif, M., Lee, J., Soulé, R., Wang, H., Caşcaval, C., McKeown, N., Foster, N.: p4v: Practical verification for programmable data planes. In: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. p. 490–503. SIGCOMM '18 (2018). https://doi.org/10.1145/3230543.3230582
12. Lööw, A.: Lutsig: a verified Verilog compiler for verified circuit development. In: Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 46–60. CPP 2021 (2021). https://doi.org/10.1145/3437992.3439916
13. Lööw, A., Myreen, M.O.: A proof-producing translator for Verilog development in HOL. In: 2019 IEEE/ACM 7th International Conference on Formal Methods in Software Engineering (FormaliSE). pp. 99–108 (2019). https://doi.org/10.1109/FormaliSE.2019.00020
14. Lundberg, D., Guanciale, R., Dam, M.: Proof-producing symbolic execution for P4. In: Verified Software. Theories, Tools and Experiments. pp. 70–83 (2025). https://doi.org/10.1007/978-3-031-86695-1_5
15. Myreen, M.O., Owens, S.: Proof-producing synthesis of ML from higher-order logic. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. p. 115–126. ICFP '12 (2012). https://doi.org/10.1145/2364527.2364545
16. Netflix, Inc.: Netflix-recommended internet speeds (2025), https://help.netflix.com/en/node/306
17. Nötzli, A., Khan, J., Fingerhut, A., Barrett, C., Athanas, P.: P4pktgen: Automated test case generation for P4 programs. In: Proceedings of the Symposium on SDN Research. pp. 1–7 (2018). https://doi.org/10.1145/3185467.3185497
18. P4 Language Consortium: Behavioral model (bmv2) (2025), https://github.com/p4lang/behavioral-model
19. Peterson, R., Campbell, E.H., Chen, J., Isak, N., Shyu, C., Doenges, R., Ataei, P., Foster, N.: P4Cub: A little language for big routers. In: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 303–319 (2023). https://doi.org/10.1145/3554345
20. Åman Pohjola, J., Syeda, H.T., Tanaka, M., Winter, K., Sau, T.W., Nott, B., Ung, T.T., McLaughlin, C., Seassau, R., Myreen, M.O., Norrish, M., Heiser, G.: Pancake: Verified systems programming made sweeter. In: Proceedings of the 12th Workshop on Programming Languages and Operating Systems. p. 1–9. PLOS '23 (2023). https://doi.org/10.1145/3623759.3624544
21. Shahbaz, M., Choi, S., Pfaff, B., Kim, C., Feamster, N., McKeown, N., Rexford, J.: Pisces: A programmable, protocol-independent software switch. In: Proceedings

of the 2016 ACM SIGCOMM Conference. p. 525–538. SIGCOMM '16 (2016). https://doi.org/10.1145/2934872.2934886
22. SONiC Foundation: PINS infrastructure (2025), https://github.com/sonic-net/sonic-pins
23. Stoenescu, R., Dumitrescu, D., Popovici, M., Negreanu, L., Raiciu, C.: Debugging P4 programs with Vera. In: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. pp. 518–532 (2018). https://doi.org/10.1145/3230543.3230548
24. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A., Owens, S., Norrish, M.: The verified CakeML compiler backend. Journal of Functional Programming **29** (2019). https://doi.org/10.1017/S0956796818000229
25. The P4 Language Consortium: $P4_{16}$ language specification (2024), https://p4.org/wp-content/uploads/2024/10/P4-16-spec-v1.2.5.html
26. Wang, Q., Pan, M., Wang, S., Doenges, R., Beringer, L., Appel, A.W.: Foundational verification of stateful P4 packet processing. In: 14th International Conference on Interactive Theorem Proving (ITP 2023). Schloss-Dagstuhl-Leibniz Zentrum für Informatik (2023). https://doi.org/10.4230/LIPIcs.ITP.2023.32
27. Wang, S., Pan, M., Appel, A.W.: Comprehensive verification of packet processing (2024). https://doi.org/10.48550/arXiv.2412.19908
28. Wiles, K.: Pktgen - traffic generator powered by DPDK (2025), https://github.com/pktgen/Pktgen-DPDK
29. Zoom Communications, Inc.: Zoom system requirements: Windows, macOS, Linux (2025), https://support.zoom.com/hc/en/article?id=zm_kb&sysparm_article=KB0060748