Compilomorphic Fuzzing: Turning a Compiler Against Itself

Vasileios Klimis
v.klimis@qmul.ac.uk
Queen Mary University of London
UK

Abstract

Compilers for structured data pipelines, often specified using domain-specific languages such as P4, are notoriously difficult to validate. We introduce *compilomorphic fuzzing*, a new validation approach that turns a compiler into its own test oracle via structure-preserving transformations over programs and structured inputs. The core idea is to generate pairs of semantically equivalent but structurally distinct inputs (e.g., packets) and programs (e.g., pipeline specifications), and then assert that the compiler's observable behaviour remains invariant. Any deviation exposes a deep semantic bug. We present the formal basis for this approach, compilomorphism, and a prototype fuzzing framework that operationalises it, establishing a practical, model-free foundation for validating compilers in critical DSL domains.

CCS Concepts

• **Software and its engineering → Software testing and debugging**; *Retargetable compilers.*

Keywords

Compiler Testing, Fuzzing, Test Oracle, P4, Domain-Specific Languages

ACM Reference Format:
Vasileios Klimis. 2026. Compilomorphic Fuzzing: Turning a Compiler Against Itself. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 05–09, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3803437.3805569

1 Introduction

Modern software systems increasingly rely on domain-specific languages (DSLs) to define behaviour for structured data pipelines, from network packet processing (P4) to graphics rendering (shaders) and data queries (SQL). Validating the compilers for these DSLs is a critical software engineering challenge. Bugs in these compilers can silently corrupt data, violate security policies, or cause system failures, yet their detection remains difficult with traditional methods.

This challenge is particularly acute in network programming with P4 [4]. P4 programs define intricate transformations on packet data, and their compilers must target a diverse range of hardware and software backends. Bugs in P4 compilers can lead to network misrouting or security vulnerabilities. Traditional validation approaches, such as comparing compiler output against a "golden" reference interpreter, are often impractical due to the state-space explosion of possible inputs and the difficulty of constructing and maintaining correct reference models for multiple, complex backends.

To address this, we propose *compilomorphic fuzzing*, a validation methodology that effectively *turns a compiler against itself*. Instead of relying on an external reference model, it leverages a DSL's own semantics to create test oracles from *structure-preserving transformations*. The core insight is that a compiler's treatment of one program variant can be used to judge its correctness on another, semantically equivalent variant. This is underpinned by the principle of *compilomorphism*: the formal definition of equivalence relations over both programs and their structured inputs.
This paper introduces compilomorphism as a validation methodology for compilers of structured data pipelines, grounded in formal equivalence relations and demonstrated via an executable prototype:

- **The Concept:** A semantics-driven oracle based on structure-preserving transformations over programs and structured inputs.
- **The Formalism:** A core framework of packet and program equivalence relations that characterise compilomorphic variants.
- **The System:** A prototype fuzzing pipeline that operationalises compilomorphism and detects injected semantic faults.

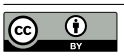
We argue that compilomorphism provides a principled and practical foundation for scalable, model-free validation of compilers in critical DSL domains.

2 Background: P4 as a Motivating Example

While the principle of compilomorphism is general, we ground our framework in the domain of P4 [4], a premier example of a DSL for structured data pipelines. In this context, *inputs* are data packets and *programs* are pipeline specifications. P4 programs define how network devices process these packets through a series of programmable stages, as illustrated conceptually in Figure 1. The challenge is that P4 compilers must correctly translate this logic for a wide range of hardware and software backends with diverse capabilities – from behavioural simulators [3] and software switches [1] to FPGAs [10] and specialised ASICs [5, 9].

The model is fundamentally based on manipulating structured data. A key component is the *Parser*, which reads raw input data (a packet) and extracts values into well-defined, typed header structures (e.g., an `IPv4Header` struct containing fields for addresses and protocol type).

The core of the pipeline logic resides in a series of control blocks containing *Match-Action Tables*. These tables implement a rule-based processing model:



FSE Companion '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2636-1/2026/07
https://doi.org/10.1145/3803437.3805569

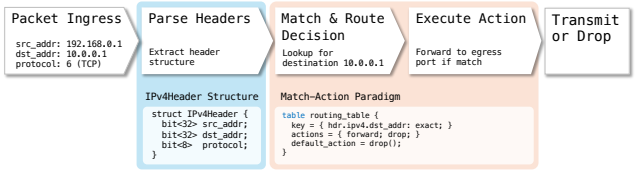


Figure 1: Conceptual stages of a P4 data pipeline: parsing inputs into structured headers, applying match-action table logic, and modifying headers before output.

- *Match:* The values of one or more header fields from the parsed packet are used as a key to look up an entry in a table.
- *Action:* Each table entry is associated with an action, which is a block of imperative code. If the key matches, the corresponding action is executed. Actions can modify header fields, update metadata, or make a final decision about the packet's fate (e.g., forward it to a specific output port or drop it).

This staged model of parsing structured inputs and applying rule-based transformations is central to why compiler validation is challenging. A compiler must correctly translate these interdependent stages into efficient code for diverse targets while preserving the exact semantics of the data transformations. At the same time, this structured computation model naturally induces rich equivalence classes over both packets and programs, which our compilomorphism relations (Section 3) are designed to exploit.

3 The Compilomorphism Framework

We formalise compilomorphism by modelling the observable behaviour of a compiled program as a transformation function $\mathcal{T} : \beta \times pg \times pk \to (pk', port)$, where β is a compiler backend, pg a source program, and pk an input packet.

Two executions are observationally equivalent if they produce the same egress decision and agree on forwarding-relevant output state. We write $(pk'_1, port_1) \approx (pk'_2, port_2)$ iff $port_1 = port_2$ and $\pi_{\text{Fwd}}(pk'_1) = \pi_{\text{Fwd}}(pk'_2)$. Here π_{Fwd} denotes projection onto the forwarding-relevant packet fields (i.e., fields that influence subsequent routing or delivery semantics). The compilomorphism principle induces three distinct forms of self-consistency checks that serve as test oracles:

- *Cross-Backend Consistency:* For a given pair (pg, pk), any two backends β_1, β_2 should produce equivalent outputs: $\mathcal{T}(\beta_1, pg, pk) \approx \mathcal{T}(\beta_2, pg, pk)$.
- *Cross-Packet Consistency:* For a given (pg, β), two compilomorphic packets $pk_1 \simeq_{pg} pk_2$ should produce equivalent outputs.
- *Cross-Program Consistency:* For a given (pk, β), two compilomorphic programs $pg_1 \cong_{pk} pg_2$ should produce equivalent outputs.

The core of our framework lies in formalising the latter two relations over structured inputs and programs.

While we use P4 terminology for clarity, these relations instantiate general patterns common to many structured data pipelines, which we characterise as systems with (a) well-defined input schemas, (b) staged, rule-based processing, and (c) transformations that must preserve external semantics.

3.1 Packet Compilomorphism

Often, many parts of a complex input are irrelevant to a specific program's logic. Two different input packets should be treated identically if they do not differ in any way that the program cares about. We formalise this as packet compilomorphism.

Definition 3.1 (Packet Compilomorphism). Let $\text{Rel}(pg)$ be the set of packet fields whose values may be read by pg and can affect observable behaviour (e.g., table keys, guards, and action expressions that influence forwarding-relevant outputs), as computed by a conservative IR analysis. Two packets pk_1 and pk_2 are compilomorphic under pg (denoted $pk_1 \simeq_{pg} pk_2$) iff their projections onto relevant fields are equal: $\pi_{\text{Rel}(pg)}(pk_1) = \pi_{\text{Rel}(pg)}(pk_2)$.

Oracle expectation. For compilomorphic packets, a correct compiler backend preserves observable behaviour: $pk_1 \simeq_{pg} pk_2 \Rightarrow \mathcal{T}(\beta, pg, pk_1) \approx \mathcal{T}(\beta, pg, pk_2)$.

3.2 Program Compilomorphism

Conversely, two structurally different programs might implement the exact same logic for a given set of inputs. A correct compiler should produce the same behaviour for both. We formalise this as program compilomorphism.

Definition 3.2 (Program Compilomorphism). Two programs pg_1 and pg_2 are compilomorphic for a packet pk (denoted $pg_1 \cong_{pk} pg_2$) iff pg_2 is obtained from pg_1 by a semantics-preserving transformation whose preconditions hold for pk (e.g., reordering independent tables, or inserting redundant logic not triggered by pk).

Oracle expectation. $pg_1 \cong_{pk} pg_2 \Rightarrow \mathcal{T}(\beta, pg_1, pk) \approx \mathcal{T}(\beta, pg_2, pk)$.

3.3 Illustrative Example

We illustrate program compilomorphism using a transformation that produces a structurally distinct program while preserving behaviour for a given packet pk.

Seed. Let pk be a packet with $DstIP = A$ and $SrcIP = B$. Let pg_1 be a program whose forwarding logic is:

> match on $DstIP$; if $DstIP = A$ then forward to port $P1$, otherwise drop.

Transformation: inserting redundant logic. We construct pg_2 from pg_1 by inserting a redundant match-action table before the routing logic. The table matches on $DstIP$ against a sentinel value S and performs a no-op action. The transformation's precondition is that $pk.DstIP \neq S$.

Since the redundant table never matches for pk, it has no effect on forwarding-relevant state. Therefore, pg_2 is obtained from pg_1 by a semantics-preserving transformation whose preconditions hold for pk, i.e., $pg_1 \cong_{pk} pg_2$.

Oracle check. A correct compiler backend must preserve this equivalence: $\mathcal{T}(\beta, pg_1, pk) \approx \mathcal{T}(\beta, pg_2, pk)$. If compiling pg_2 results in a different egress decision (e.g., drop), we have detected a semantic compiler bug.

4 Compilomorphic Fuzzing

We operationalise compilomorphism through a fuzz testing framework that uses the compiler's own semantic self-consistency across compilomorphic variants as a live test oracle (Figure 2).

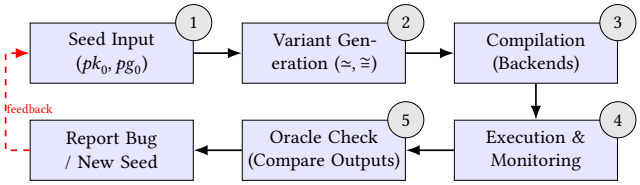


Figure 2: The compilomorphic fuzzing workflow. Variants are generated (2), compiled (3), and executed (4), with outputs compared against the oracle (5). A feedback loop can use results to generate new seeds.

Algorithm 1 Compilomorphic Fuzzing Logic

procedure FuzzCompiler(pk_0, pg_0, B)
  $TestCorpus \leftarrow \{(pk_0, pg_0)\}$
  $ResultsCache \leftarrow \emptyset$ ▷ Cache for (pk, pg, beta) -> output
  loop
    $(pk_s, pg_s) \leftarrow$ select_from($TestCorpus$)
    $(pk_m, pg_m) \leftarrow$ generate_variant(pk_s, pg_s)
    if (pk_m, pg_m) is new **then**
      $TestCorpus.add((pk_m, pg_m))$
      for all $\beta \in B$ **do** ▷ Execute and cache results
        $output \leftarrow$ execute(β, pg_m, pk_m)
        $ResultsCache[(pk_m, pg_m, \beta)] \leftarrow output$
  Oracle check
    $(pk_1, pg_1), (pk_2, pg_2) \leftarrow$ SelectPair($TestCorpus$)
    for all $\beta_1, \beta_2 \in B$ **do**
      $out_1 \leftarrow ResultsCache[(pk_1, pg_1, \beta_1)]$
      $out_2 \leftarrow ResultsCache[(pk_2, pg_2, \beta_2)]$
      if $out_1 \neq out_2$ **then**
        Report Bug: Inconsistency found!
        return

The core logic of the fuzzer, outlined in Algorithm 1, follows the workflow shown in Figure 2. It iteratively generates new test cases and checks for inconsistencies. The key stages are:

Seeding and Variant Generation. The process starts with a seed program pg_0 and packet pk_0. The *Variant Generation* engine (Figure 2, Step 2) is the heart of the fuzzer. It applies semantics-preserving mutations to generate new, equivalent variants guided by the compilomorphism relations:

- *To satisfy Packet Compilomorphism (≃):* Based on a static analysis of the program's IR, the engine creates new packets by modifying fields not relevant to the program's logic (e.g., perturbing unused header fields).
- *To satisfy Program Compilomorphism (≅):* The engine applies a predefined set of sound, semantics-preserving transformations, such as reordering independent tables or adding redundant logic.

Execution and Oracle Check. Each generated program variant is compiled for all backends under test (Step 3) and executed with its corresponding packet while outputs are monitored (Step 4). The *Oracle Check* (Step 5) is where the compiler is tested against itself: the framework compares the outputs of any two compilomorphic pairs. A mismatch reveals an inconsistency where the compiler failed to preserve the expected semantics and is reported as a likely bug.

Formal Grounding and Feasibility. We use an executable Alloy model to validate the internal consistency of the compilomorphism framework (e.g., variant-generation constraints). The model is not a compiler oracle, but a design-time sanity check that can generate small compilomorphic witnesses.

Prototype Implementation. We implemented a minimal end-to-end prototype that operationalises the compilomorphism framework. The system performs P4 IR extraction, generates packet- and program-compilomorphic variants, executes them via a mock runner that models match-action evaluation and table application order, and applies the self-consistency oracle induced by compilomorphic variants with disparity-based classification.

To validate bug detection, we injected two classes of semantic faults (control-flow and data-plane); both were detected and correctly classified by the oracle. The IR analysis is conservative: when field relevance is uncertain, the prototype marks the field as relevant, preserving soundness of generated variants. The prototype is available as a supplementary artefact alongside the Alloy model on Zenodo DOI 10.5281/zenodo.18332133.

Disparity-Based Bug Classification. When the oracle check fails ($out_1 \neq out_2$), the framework computes a *disparity vector* that characterises the observable difference. We instantiate disparity as $\Delta = \langle port\_diff, header\_diff \rangle$, where *port_diff* indicates whether the egress decisions differ and *header_diff* summarises differences over forwarding-relevant output fields.

This representation distinguishes control-flow miscompilations ($\langle$true, false$\rangle$) from data-plane miscompilations ($\langle$false, true$\rangle$), providing lightweight diagnostic information in addition to a pass/fail oracle.

5 Related Work

Our work connects research on compiler testing, differential and metamorphic testing, translation validation, and domain-specific compiler validation, with a focus on semantics-based test oracles.

Differential and Randomised Compiler Testing. Differential testing compares multiple executions of the same program to expose inconsistencies [17]. Equivalence Modulo Inputs (EMI) applies this principle to compiler validation by generating semantics-preserving program variants and checking for mismatches on a given input [13]. Athena extends EMI with guided stochastic mutation to discover deep miscompilation bugs [14]. Random program generators such as CSmith have been highly effective for finding bugs in C compilers [23], while coverage-guided fuzzers such as AFL [24] prioritise syntactic diversity but typically lack strong semantic oracles. Empirical studies show that equivalence-based methods are among the most effective at discovering deep semantic bugs [6].

Our key contribution over this body of work is the formalisation and systematic exploration of a **joint equivalence space** across both programs and structured inputs. While prior methods typically vary one or the other, compilomorphism's dual-pronged approach is designed for domains like P4 where input validity is tightly coupled with program logic, making the simultaneous consideration of both essential for generating meaningful test cases.

Metamorphic Testing. Metamorphic testing alleviates the oracle problem by checking whether expected relations hold between

the outputs of multiple executions under systematic input transformations [7]. Compilomorphism is related in spirit but differs in scope and operationalisation. Rather than varying only inputs, it treats programs and structured inputs as first-class objects and mutates both via semantics-preserving transformations, using a domain-informed notion of observable behaviour and disparity-based triage.

Translation Validation and Compiler Correctness. Translation validation checks the correctness of individual compiler runs by verifying semantic equivalence between source and target representations [18, 20]. Verified compilers such as CompCert provide stronger guarantees by construction [15], but require full formal models of the compiler and target. Compilomorphism is complementary: it treats the compiler as a black box and validates semantic preservation empirically via structured equivalence testing.

P4-Specific Validation and Testing. The P4 ecosystem includes formal semantics and verification tools such as Petr4 [8], p4v [16], P4Cub [19], and P4K [11], which reason about correctness of individual programs. Testing frameworks such as P4Testgen [21] generate high-coverage packet tests from program semantics, while network-level validation systems such as Meissa [25] validate end-to-end data-plane behaviour.

Several tools directly target P4 compiler correctness. Gauntlet applies random program generation and translation validation to discover miscompilations [22], while P4Fuzz uses generation-based fuzzing tailored to P4 [2]. These approaches either rely on backend-specific models or use weak oracles focused on crashes and invalid behaviour. In contrast, compilomorphism provides a model-free semantic oracle based on structure-preserving transformations over programs and structured inputs.

Domain-Specific Compiler Validation. Beyond P4, DSL compiler validation has been explored in other domains, including GPU and intermediate-representation pipelines [12]. Industrial DSLs such as Broadcom's NPL further illustrate the growing need for semantics-aware validation techniques in programmable networking platforms.

Together, these works motivate the need for principled, semantics-grounded validation oracles that scale across structured inputs, heterogeneous backends, and domain-specific execution models.

6 Implications and Vision

Compilomorphic fuzzing reframes compiler validation. By using the compiler's expected behaviour on one set of inputs to oracle its behaviour on an equivalent but different set, we effectively turn the compiler against itself. This perspective has several implications for software testing and verification.

Semantics-Guided Fuzzing. Traditional grammar-based fuzzers excel at exploring syntactic variation but struggle to generate semantically meaningful programs. Compilomorphism instead guides mutation via semantics-preserving transformations, enabling targeted exploration of compiler logic and exposing incorrect-computation bugs rather than merely crashes.

Validation for Heterogeneous Targets. Modern DSL compilers target increasingly heterogeneous backends (CPUs, GPUs, FPGAs, ASICs), making behavioural consistency difficult to validate. By treating each backend as an instance of the transformation function $\mathcal{T}$, compilomorphism provides a principled basis for differential testing across this entire target space.

Broader Applicability. Although P4 is our motivating example, the underlying pattern is common across DSLs that define transformations over structured data.

- *Graphics:* Shader compilers must preserve visual semantics across diverse GPUs. Compilomorphic variants can be generated from mathematically equivalent lighting or vertex transformations.
- *Database Queries:* Query optimisers transform declarative SQL into execution plans. Logically equivalent queries (e.g., via join reordering) form compilomorphic program variants and must produce identical result sets.

These domains share the same validation challenge: ensuring that semantics are preserved across aggressive optimisation and diverse execution platforms.

7 Discussion: Scope and Limitations

Compilomorphism is a general principle, but our current instantiation has clear scope boundaries that motivate future work.

Stateful Operations. Our approach currently targets stateless, deterministic transformations. Extending it to stateful constructs (e.g., P4 counters or registers) would require defining equivalence over sequences of inputs and the resulting evolution of program state.

Transformation Soundness. The effectiveness of our fuzzing approach depends on sound semantics-preserving transformations. While simple cases (e.g., reordering independent tables) are easy, defining and validating richer program refactorings is a key challenge. Automating the discovery of such transformations is an important research direction.

Non-Functional Properties. Our framework targets only semantic correctness, not non-functional properties like performance or resource usage. Two compilomorphic variants might behave identically but have vastly different efficiency profiles, an aspect our oracle does not currently address.

Shared-Mode Failures. Like all differential testing, compilomorphism detects inconsistencies, not absolute correctness. A compiler could produce the same wrong output for two variants. Detecting such shared-mode failures requires a complementary oracle, such as a reference model.

8 Conclusion

We introduced compilomorphic fuzzing, a validation methodology that turns a compiler into its own semantic oracle via structure-preserving transformations over programs and inputs. We formalised packet and program equivalence relations, showed how they induce a model-free semantic oracle, and demonstrated feasibility via an executable prototype that detects injected semantic faults.

We argue that compilomorphism provides a principled foundation for scalable, automated validation of DSL compilers and a unifying perspective on differential, metamorphic, and translation-validation techniques. Future work will focus on scaling the system to production P4 toolchains and evaluating its effectiveness on real-world compiler bugs.

References

[1] Adarsh Rawat. . eBPF Backend). https://github.com/p4lang/p4c/blob/main/backends/ebpf/README.md

[2] Andrei-Alexandru Agape, Madalin Claudiu Danceanu, Rene Rydhof Hansen, and Stefan Schmid. 2021. P4Fuzz: Compiler Fuzzer for Dependable Programmable Dataplanes. In *Proceedings of the 22nd International Conference on Distributed Computing and Networking (ICDCN)*.

[3] Antonin Bas. . The Reference P4 Software Switch). https://github.com/p4lang/behavioral-model

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* (2014).

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. *SIGCOMM Comput. Commun. Rev.* 43, 4 (2013).

[6] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*.

[7] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* (2018).

[8] Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. 2021. Petr4: Formal Foundations for P4 Data Planes. *Proc. ACM Program. Lang.* POPL (2021).

[9] David Franco, Eder Ollora Zaballa, Mingyuan Zang, Asier Atutxa, Jorge Sasiain, Aleksander Pruski, Elisa Rojas, Marivi Higuero, and Eduardo Jacob. 2024. A comprehensive latency profiling study of the Tofino P4 programmable ASIC-based hardware. *Computer Communications* 218 (2024).

[10] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. 2019. The P4->NetFPGA Workflow for Line-Rate Packet Processing. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.

[11] Ali Kheradmand and Grigore Rosu. 2018. P4K: A Formal Semantics of P4 and Applications. arXiv:1804.01468 [cs.NI] https://arxiv.org/abs/1804.01468

[12] Vasileios Klimis, Jack Clark, Alan Baker, David Neto, John Wickerson, and Alastair F. Donaldson. 2023. Taking Back Control in an Intermediate Representation for GPU Computing. *Proc. ACM Program. Lang.* 7, POPL (2023).

[13] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[14] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[15] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. SEE, Toulouse, France. https://inria.hal.science/hal-01238879

[16] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. p4v: practical verification for programmable data planes. In *Proceedings of the ACM SIGCOMM*.

[17] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal*, (1998).

[18] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Programming Language Design and Implementation (PLDI)*.

[19] Rudy Peterson, Eric Hayden Campbell, John Chen, Natalie Isak, Calvin Shyu, Ryan Doenges, Parisa Ataei, and Nate Foster. 2023. P4Cub: A Little Language for Big Routers *(CPP 2023)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3573105.3575670

[20] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[21] Fabian Ruffy, Jed Liu, Prathima Kotikalapudi, Vojtech Havel, Hanneli Tavante, Rob Sherwood, Vladyslav Dubina, Volodymyr Peschanenko, Anirudh Sivaraman, and Nate Foster. 2023. P4Testgen: An Extensible Test Oracle For P4. In *Proceedings of the ACM SIGCOMM*.

[22] Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. 2020. Gauntlet: Finding Bugs in Compilers for Programmable Packet Processing. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[23] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[24] Michał Zalewski. [n. d.]. American Fuzzy Lop. https://github.com/google/AFL. Accessed: September 11, 2025.

[25] Naiqian Zheng, Mengqi Liu, Ennan Zhai, Hongqiang Harry Liu, Yifan Li, Kaicheng Yang, Xuanzhe Liu, and Xin Jin. 2022. Meissa: scalable network testing for programmable data planes. In *Proceedings of the ACM SIGCOMM*.