Unifying eBPF across Platforms: Formal Semantics, Conformance Testing, and Specifications

Yanze Li
University of British Columbia
yanzeli@cs.ubc.ca

Reto Achermann
Technical University of Munich
achreto@cit.tum.de

Ivan Beschastnikh
University of British Columbia
bestchai@cs.ubc.ca

Alexander J. Summers
University of British Columbia
alex.summers@ubc.ca

Abstract

eBPF is no longer a single-platform technology. It runs in the Linux kernel, on Windows, in user space, on microcontrollers, and in blockchain virtual machines, on independently built runtimes. The IETF ISA standard, RFC 9669, pins down the core instructions but leaves out features that real programs depend on, such as helper functions and maps. We are building an executable formal semantics for eBPF in F* that explicitly distinguishes cross-platform and platform-specific behaviors. Our semantics passes the BPF conformance test suite on par with uBPF, bpftime, Linux, and Windows, and we are extending it beyond the core ISA to other shared features the RFC omits. Using Meta-F* metaprogramming, we can also generate prose specifications in structured English that provably match the model. We envision this semantics as a practical foundation for a uniform, trustworthy eBPF across platforms.

CCS Concepts: • **Software and its engineering → Semantics**; **Correctness**.

Keywords: eBPF, formal semantics, conformance testing, cross-platform portability, specification

ACM Reference Format:
Yanze Li, Reto Achermann, Ivan Beschastnikh, and Alexander J. Summers. 2026. Unifying eBPF across Platforms: Formal Semantics, Conformance Testing, and Specifications. In *4th Workshop on eBPF and Kernel Extensions (eBPF '26), September 29–October 2, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3837779.3838165

1 Problem and Motivation

eBPF began as a mechanism for safely extending the Linux kernel, and it is no longer Linux-specific. eBPF runtimes now target many platforms: Windows [9], user space [6, 18], IoT microcontrollers [17], and blockchain virtual machines [1]. Many are under active development and overlap in unexpected ways; uBPF, for example, ships as part of the Windows eBPF runtime [6, 9]. Built independently and for different purposes, these runtimes disagree on what eBPF *means*.

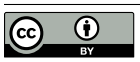
Some of the disagreements in these runtimes are by design. The Linux verifier rewrites the bytecode as it verifies it: it remaps context accesses to kernel-structure offsets, and it inserts guards that enforce shared semantics, for example so that the signed-division overflow `INT_MIN s/ -1` returns `INT_MIN` instead of faulting. Windows instead runs PREVAIL, a user-space verifier [3] that only accepts or rejects, and leaves such transformation to a separate step that compiles the verified bytecode to C through `bpf2c`. The program that runs need not be the bytecode that was written; what it becomes depends on the platform's verifier and toolchain.

An IETF standard for the instruction set now exists [13], but it does not fully close the gap. It is natural-language prose with under-specified corners, and its process lags behind the implementations. For example, the defined result of the `INT_MIN s/ -1` overflow above, which runtimes already implement, was deferred because RFC 9669 had been frozen [11]. The standard's scope is also narrow: it defines instruction encodings but not the execution model a runtime needs to run a program, such as the calling convention and register roles, the stack and frame pointer, the program-entry context, and the helper and map interfaces [13]. No artifact states, executably and in one place, what an eBPF program does; each runtime is its own de facto specification.

Such gaps between implementations and the standard weaken eBPF's compile-once-run-everywhere (CO-RE) portability and make eBPF's safety argument platform-specific [4].

2 An Executable Cross-Platform Semantics

Prior formal semantics for eBPF each target a single platform: microcontroller rBPF [14], Solana [16], and Linux [15]. Our semantics, written in F* [12], is cross-platform and designed to be executable: it can be used as an interpreter for eBPF bytecode. We exploit F*'s rich toolchain: SMT-backed proof automation, Meta-F* metaprogramming for spec generation, and extraction to OCaml and C.



This work is licensed under a Creative Commons Attribution 4.0 International License.
eBPF '26, Prague, Czech Republic
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2911-9/26/09
https://doi.org/10.1145/3837779.3838165

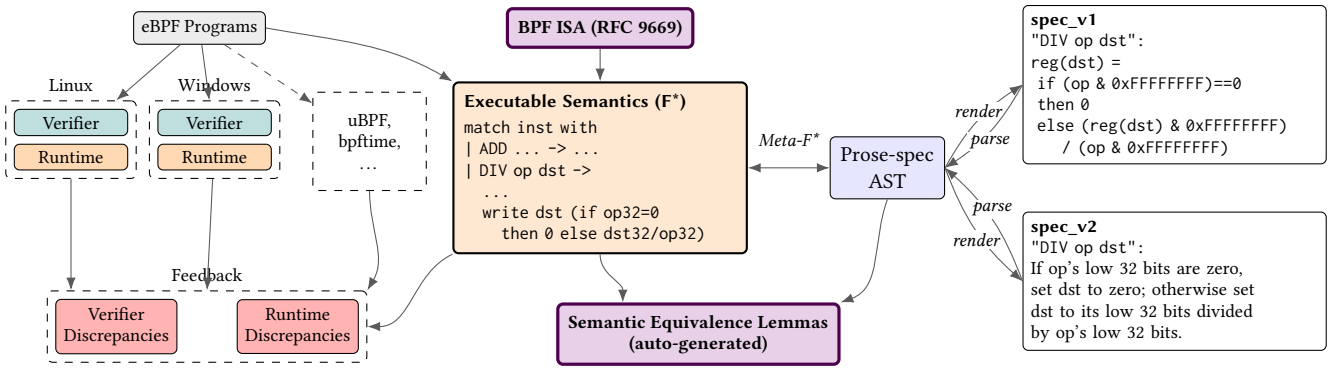


Figure 1. Our executable F* semantics formalizes the BPF ISA (RFC 9669) and features shared across runtimes. *Left (conformance testing):* eBPF programs run through each platform's verifier and runtime and through our semantics, exposing discrepancies. *Right (faithful prose spec):* Meta-F* maps the semantics to a prose-spec AST that renders to (and parses back from) prose forms such as `spec_v1` and `spec_v2`; an auto-generated lemma certifies the AST is equivalent to our semantics.

Much of eBPF's semantics is shared across platforms: runtimes agree on more than the ISA the RFC defines, namely the calling convention, register roles, stack, and frame pointer. In contrast, the helper and `kfunc` sets, the map types, the memory model, and the stack size differ. We capture the shared semantics in a platform-independent core and expose platform-specific operations through an F* typeclass. This makes shared and unshared semantics explicit and machine-checked, rather than buried in C and Rust implementations.

3 What Our Semantics Enables

Figure 1 overviews how our semantics enables conformance testing and faithful prose specifications.

Conformance testing. We run the BPF conformance test suite [7] on our executable semantics and on other target runtimes, comparing the resulting register and memory state. Our semantics passes the suite on par with uBPF [6], bpftime [18], Linux, and Windows [9]. This validates the shared core against the standardized ISA. The method is runtime-agnostic, and covering a new target such as Solana's SBPF [1] needs only a thin driver.

Conformance beyond the RFC. The BPF conformance test suite covers only the instructions in RFC 9669. However, eBPF programs in practice depend on more features the RFC puts out of scope, notably helper functions and maps. For example, both Windows and bpftime converge on a Linux-compatible subset of helper functions. We are extending both our semantics and the conformance suite to cover these shared features, where cross-platform divergence is far more likely than in the core ISA.

Conformance of verifiers and their guarantees. Verifiers make conformance subtler. Linux rewrites the bytecode during verification, whereas Windows's PREVAIL does not. Comparing runtimes alone can therefore mislead, hence we are also modeling the verifiers on top of our semantics, capturing their accept/reject decisions and any rewriting. This lets us compare verification results faithfully, pinpoint the root cause when implementations disagree, and evaluate whether each platform's verifier enforces the same safety guarantees, connecting our work to eBPF verifier correctness [4] and differential testing from specifications [10].

Faithful prose from formalism. Specifications are usually read as prose. Frameworks such as Sail [2] are useful for generating readable documentation from an ISA model. However, such tools do not guarantee the rendered prose corresponds to the model. Our goal is therefore to provide stronger guarantees when generating prose specifications. We use Meta-F* metaprogramming [8] to (i) build a core AST capturing the structure of the prose specification and confine the generated prose to stylized, machine-checkable English and (ii) emit, per instruction, a lemma that the prose meaningfully corresponds to our F* semantics. This extends IETF practice: an `ebpf-check` script derives an instruction table from the RFC and validates object files against the defined bytecode *syntax* [5]. Our work, by contrast, generates prose specifications that provably track our validated semantics.

4 Discussion and Vision

Our executable semantics covers all RFC 9669 instructions except atomic operations, and is conformance-tested against Linux, Windows, uBPF, and bpftime. Extensions to helpers and maps, verifier models, and faithful prose generation are in progress. Our aim is a single executable definition of eBPF that runtimes, verifiers, and the prose standard can all be checked against, keeping the implementations and the standard aligned.

Acknowledgments

This work was supported by NSERC Discovery Grants (RGPIN-2025-06826, RGPIN-2025-06888, and RGPIN-2025-04203). We thank Hugh Coleman, Dennis Gu, and Jiayin Kralik for their contributions to the project.

References

[1] Anza. [n. d.]. SBPF: Solana BPF Virtual Machine. https://github.com/anza-xyz/sbpf. Accessed June 2026.

[2] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 71:1–71:31.

[3] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 1069–1084.

[4] Kaiming Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. 2025. SoK: Challenges and Paths Toward Memory Safety for eBPF. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 848–866.

[5] IETF BPF Working Group. [n. d.]. eBPF Documentation and Validation Tools (`ebpf-check`). https://github.com/ietf-wg-bpf/ebpf-docs. Accessed June 2026.

[6] IO Visor Project. [n. d.]. uBPF: Userspace eBPF VM. https://github.com/iovisor/ubpf. Accessed June 2026.

[7] Alan Jowett. [n. d.]. `bpf_conformance`: Tests Measuring BPF Runtime Conformance to the ISA. https://github.com/Alan-Jowett/bpf_conformance. Accessed June 2026.

[8] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hriţcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. 2019. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *European Symposium on Programming (ESOP)*. Springer, 30–59.

[9] Microsoft. [n. d.]. eBPF for Windows. https://github.com/microsoft/ebpf-for-windows. Accessed June 2026.

[10] Nikitha Rao, Elizabeth Gilbert, Harrison Green, Tahina Ramananandro, Nikhil Swamy, Claire Le Goues, and Sarah Fakhoury. 2024. DiffSpec: Differential Testing with LLMs using Natural Language Specifications and Code Artifacts. *arXiv preprint arXiv:2410.04249* (2024).

[11] Yonghong Song, Dave Thaler, and Alexei Starovoitov. 2024. Document Some Special `sdiv`/`smod` Operations. Discussion on the IETF BPF mailing list. https://mailarchive.ietf.org/arch/msg/bpf/RIsU3BSetPq3LrX7rc4qFb4bEJw/.

[12] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 256–270.

[13] Dave Thaler. 2024. BPF Instruction Set Architecture (ISA). RFC 9669, Internet Engineering Task Force (IETF). https://www.rfc-editor.org/info/rfc9669.

[14] Shenghao Yuan, Frédéric Besson, Jean-Pierre Talpin, Samuel Hym, Koen Zandberg, and Emmanuel Baccelli. 2022. End-to-end Mechanized Proof of an eBPF Virtual Machine for Micro-controllers. In *International Conference on Computer Aided Verification (CAV)*. Springer, 293–316.

[15] Shenghao Yuan, Yazhou Tang, Tianci Cao, Frédéric Besson, Jean-Pierre Talpin, and Mingshuai Chen. 2026. Formalizing the Linux eBPF Core ISA: A Mechanized Operational Semantics and Its Real-World Applications. https://github.com/shenghaoyuan/linux-ebpf-semantics. To appear at OOPSLA 2026. Accessed June 2026.

[16] Shenghao Yuan, Zhuoruo Zhang, Jiayi Lu, David Sanan, Rui Chang, and Yongwang Zhao. 2025. A complete formal semantics of ebpf instruction set architecture for solana. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 1–27.

[17] Koen Zandberg and Emmanuel Baccelli. 2020. Minimal Virtual Machines on IoT Microcontrollers: The Case of Berkeley Packet Filters with rBPF. In *2020 9th IFIP International Conference on Performance Evaluation and Modeling in Wireless Networks (PEMWN)*. IEEE, 1–6.

[18] Yusheng Zheng, Tong Yu, Yiwei Yang, Yanpeng Hu, Xiaozheng Lai, Dan Williams, and Andi Quinn. 2025. Extending Applications Safely and Efficiently. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 557–574.