A complete formal semantics of eBPF instruction set architecture for Solana

SHENGHAO YUAN, Zhejiang University, China
ZHUORUO ZHANG, Zhejiang University, China
JIAYI LU, Zhejiang University, China
DAVID SANAN, InfoComm Technology Cluster, Singapore Institute of Technology, Singapore
RUI CHANG, Zhejiang University, China
YONGWANG ZHAO, Zhejiang University, China

We present the first and most comprehensive formal semantics for the Solana eBPF bytecode language used in smart contracts on the Solana blockchain platform. Our formalization accurately captures all binary-level instructions of the Solana eBPF instruction set architecture. This semantics is structured in a small-step style, facilitating the formalization of the Solana eBPF interpreter within Isabelle/HOL. We provide a semantics validation framework that extracts an executable semantics from our formalization to test against the original implementation of the Solana eBPF interpreter. This approach introduces a novel lightweight and non-invasive method to relax the limitations of the existing Isabelle/HOL extraction mechanism. Furthermore, we illustrate potential applications of our semantics in the formalization of the main components of the Solana eBPF virtual machine.

Additional Key Words and Phrases: eBPF, ISA, Semantics, Virtual Machine, Solana Blockchain, Formal Verification, Isabelle/HOL

1 Introduction

Blockchain technology is inherently safety-critical, where even subtle issues may lead to significant consequences. To ensure safety and most importantly, provide trustworthiness of their platforms to investors/users, blockchain communities advocate for the use of formal methods, *i.e.,* the rigorous mathematical techniques aimed at proving the absence of bugs in software. The formal semantics of blockchain languages, spanning from high-level smart contract languages to low-level virtual instruction set architectures (ISAs) of blockchain virtual machines (VMs), serve as a crucial foundation for accurately describing the intricate behaviours of blockchain platforms. Existing work on formal semantics includes Ethereum Virtual Machine (EVM) [Amani et al. 2018; Cassez et al. 2023; Hildenbrandt et al. 2018; Hirai 2017; Li et al. 2019], Solidity [Jiao et al. 2020; Marmsoler and Brucker 2021], Azure Blockchain [Wang et al. 2019], and many other related formal applications, *e.g.,* verified EVM verifier [Park et al. 2018], verification of Deposit smart contract [Park et al. 2020], the move prover [Zhong et al. 2020], verified EVM block-optimization [Albert et al. 2023], and the symbolic execution tool HEVM [Dxo et al. 2024], etc.

Solana, a third-generation blockchain platform, is recognized for its high performance and lower transaction fees. It employs Linux eBPF sandboxing techniques to implement its VM for executing Solana smart contracts. Despite the entire VM [Solana-labs 2018] being developed in the memory-safe language Rust, a security review by Kudelski

Authors' Contact Information: Shenghao Yuan, Zhejiang University, China; Zhuoruo Zhang, Zhejiang University, China; Jiayi Lu, Zhejiang University, China; David Sanan, InfoComm Technology Cluster, Singapore Institute of Technology, Singapore; Rui Chang, Zhejiang University, China; Yongwang Zhao, Zhejiang University, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
Manuscript submitted to ACM

Security [Security 2019] highlights that the Rust implementation of the Solana eBPF VM remains unverified and lacks thorough code analysis and testing. Recent findings have already uncovered serious vulnerabilities within the Solana eBPF VM, including infinite loops in instruction fuel consumption [BoredPerson 2024] and call-out-of-branch issues [Solana-labs 2024]. Current research on Solana primarily focuses on detecting vulnerabilities at the smart contract level [Cui et al. 2022] and fuzzing techniques [Smolka et al. 2023]. However, to the best of our knowledge, there still remains a significant absence of formal semantics for the Solana eBPF VM, which hinders the development of a robust foundation for further formal verification.

The Solana eBPF, abbreviated as SBPF, originated from Linux eBPF but has since diverged significantly. Previous work on eBPF verification [Nelson et al. 2020; Yuan et al. 2022] has mainly focused on verified components of eBPF VMs or formalization of specific subsets of the eBPF ISA. In contrast, this paper addresses the challenge of developing a comprehensive formal semantics. To this end, we present the first complete formal semantics of the binary-level SBPF ISA. This binary-level semantics enables us to formalize the rest of the Solana VM components, *e.g.,* the SBPF assembler, disassembler, and particularly the x86-64 Just-In-Time (JIT) compiler, along with the proofs of several key properties. All formalizations and proofs presented in this paper have been mechanically verified using the Isabelle/HOL proof assistant [Nipkow et al. 2002].

1.1 Challenges

Formalizing the SBPF ISA semantics in Isabelle/HOL poses major challenges.

Inconsistent Instruction Variants. The SBPF ISA, a variant of Linux eBPF, exhibits significant differences in the behaviour of its instructions. For example, in eBPF, the 32-bit addition instruction performs unsigned addition behaviour, whereas in SBPF, it executes a 32-bit signed addition operation. Such inconsistency between eBPF and SBPF makes it difficult to refer to existing eBPF semantics [Nelson et al. 2020; Yuan et al. 2022]. Furthermore, discrepancies in the original Solana VM implementation contribute to this inconsistency: the SBPF verifier permits a version-specific instruction that the SBPF interpreter does not. This inconsistency at the source code level complicates the formalization of the SBPF ISA. In fact, we identified bugs in the Solana Rust implementation arising from incorrect instruction version checks (details in Section 7.1).

Lack of Documentation. Unlike well-established ISAs such as x86-64, ARM, and RISC-V, which have extensive documentation and formal semantics research [Armstrong et al. 2019; Dasgupta et al. 2019; Leroy 2009; Sewell et al. 2010], the SBPF ISA lacks an official manual or standardization, mainly due to its rapid evolution and frequent version iterations. Formalizing the SBPF ISA involves the non-trivial task of identifying all corner cases directly from the source code. Recently, Linux eBPF has recently proposed a draft of its standard documentation [Thaler 2024] to the IETF BPF Working Group (albeit without formal semantic support), providing a complete formal semantics for SBPF ISA would offer a solid foundation for future SBPF standardization efforts.

Complex Semantics of the Host Language (Rust). The semantics of SBPF ISA is described in accordance with the Rust implementation of the Solana interpreter, thus introducing complexities due to Rust's inherent features. For instance, the interpreter relies on intentionally-wrapped arithmetic functions to implement specific SBPF instructions. Additionally, the different semantics in basic operators between Rust and Isabelle/HOL, such as division and modulo, particularly when applied to negative numbers, significantly complicate the pursuit of a complete and faithful formalization.

Validation Gap. The original SBPF interpreter is implemented in the low-level Rust programming, whereas the formalism of SBPF ISA is modelled in the high-level Isabelle/HOL functional language leveraging the "Word" library [Dawson 2009]. Currently, there are no verified compilers that facilitate direct translation between Rust and Isabelle/HOL. As an alternative, we resort to executable semantics via the Isabelle/HOL extraction mechanism to validate the formal semantics. However, the Isabelle/HOL extraction process yields "Word" types that are notably less accessible to human readers. Another solution [Lochbihler 2018] requires modifying the Isabelle/HOL model and providing additional proofs to ensure correctness. These limitations make it challenging to validate whether our abstract formalization accurately captures the expected behaviour of the Solana interpreter.

1.2 Contributions

In this paper, we address these challenges by presenting the first and most complete semantics of the SBPF ISA for application to the formalization of the Solana VM. Specifically, we make the following contributions:

Complete Semantics of the SBPF ISA. We present the most comprehensive formal semantics of SBPF to date. Specifically, we formalize all binary-level instructions of the SBPF ISA in Isabelle/HOL, covering the entire set of 116 opcodes. These include Arithmetic Logic Unit (ALU), byte-swap, branching, memory operations, function calls, and exit behaviours.

Semantics Validation. We introduce a lightweight and non-invasive approach for validating the executable semantics generated by the Isabelle/HOL extraction mechanism. This executable semantics has undergone thorough testing against the Solana official test suite and over 100,000 automatically generated benchmarks within our validation framework. Throughout the validation progress, we successfully identified several subtle and deeply hidden inconsistencies between our Isabelle/HOL model and the original Solana interpreter.

Solana VM Formalization. Building on our binary-level formalization of the SBPF ISA, we extend our work to include the remaining components of the Solana VM. This encompasses a consistency proof for the SBPF assembler-disassembler pair, a formalization of the verifier, and a partial proof related to the JIT compilation.

Plan. The rest of the paper is organized as follows. Section 2 provides some background on BPF, eBPF, and SBPF. Section 3 outlines our approach, Section 4 proposes the semantic formalization of the full SBPF ISA. Section 5 validates the semantics. Section 6 presents a collection of semantics applications for Solana VM. Section 7 evaluates the implementation of our formal semantics and the original Solana VM. Section 8 introduces related works, Section 9 concludes, and the last section provides the data-availability statement.

2 Background

This section introduces the essential features of BPF, Linux eBPF, and Solana eBPF.

2.1 BPF and Linux eBPF

BPF [McCanne and Jacobson 1993] was initially developed to enable flexible network packet filtering by allowing users to provide BPF instructions that specify packet filter rules, which are executed directly within the kernel. This avoided costly context switching and data copying typically associated with user-space filtering. This classical BPF, also known as cBPF, is highly restrictive and limited, featuring only two registers and bytecode interpretation. Such restrictiveness becomes an obstacle in emerging scenarios that demand rich functionality and low overhead.

Modern Linux kernels now support extended BPF (eBPF) [Fleming 2017], a subsystem that extends BPF's capabilities beyond packet filtering to a wide array of applications, including kernel profiling, load balancing, and firewalling. Popular tools such as Docker, Katran [Incubator 2018], and kernel debugging utilities like Kprobes [Goswami 2005] leverage or are built directly on top of eBPF.

The Linux eBPF subsystem provides ten general-purpose 64-bit registers, R0–R9, along with a frame pointer, R10, which points to a dedicated stack memory region. It also offers eBPF-specific data structures (often referred to as eBPF maps) and a set of helper functions.

The eBPF instructions have a general format encoded in slots of 64 bits, as shown in Figure 1. A 64-bit eBPF bytecode, from least significant bit (lsb) to most significant bit (msb), consists of the following fields:

- 8-bit opcode
- 4-bit destination register and 4-bit source register
- 16-bit signed integer offset
- 32-bit signed integer immediate value

Fig. 1. Linux eBPF instruction encoding format

Since eBPF bytecode is often written by untrusted users, the kernel employs a verifier to perform a series of checks at load time. Once the verification process succeeds, the validated bytecode is either interpreted by the eBPF interpreter or further compiled into native machine code by an in-kernel, target-specific JIT compiler for optimized performance.

2.2 Solana eBPF

eBPF has been adapted for various environments, including eBPF for Windows [Microsoft 2019], the Internet of Things (IoT) operating system RIOT-OS' Femto-Containers [Zandberg et al. 2022], and most notably, the Solana eBPF VM (or SBPF for short). Solana smart contracts, typically written in languages like C or Rust, are compiled to Solana eBPF bytecode, which can be executed in either JIT compilation or interpreter mode. Solana's runtime enforces several execution constraints when running on-chain programs in the eBPF VM. By default, the SBPF VM limits the computational resources of each instruction to a specific number of compute units (CUs). The Solana runtime accumulates these compute units for all instructions within a transaction, with certain runtime operations, such as system calls, consuming a fixed number of compute units. When executing an instruction in the SBPF VM, it is serialized and passed to the VM, with the program input starting at a fixed address in the VM's memory layout.

Similar to eBPF, SBPF is a 64-bit register-based virtual machine that uses fixed-size 64-bit instructions. Its ISA derives from eBPF. SBPF consists of three primary components: a compact verifier (around 0.4k lines of code) that performs basic validation (e.g., excluding illegal opcodes), an interpreter, and an x86-64 JIT compiler for execution. Additionally, SBPF provides an assembler and disassembler to bridge between SBPF bytecode and its assembly representation.

The Solana smart contracts are often written in *e.g.,* Rust pseudo code, then these programs are translated into bytecode using a specific LLVM compiler with the support of the SBPF backend. As shown in Figure 2, if the SBPF verifier validates the provided bytecode scripts, the selected SBPF execution engine runs them. In practice, an assembler is used to generate a SBPF assembly instruction from the corresponding bytecode before the verifier checks and the consequent execution. Solana also provides a disassembler for debugging and testing.

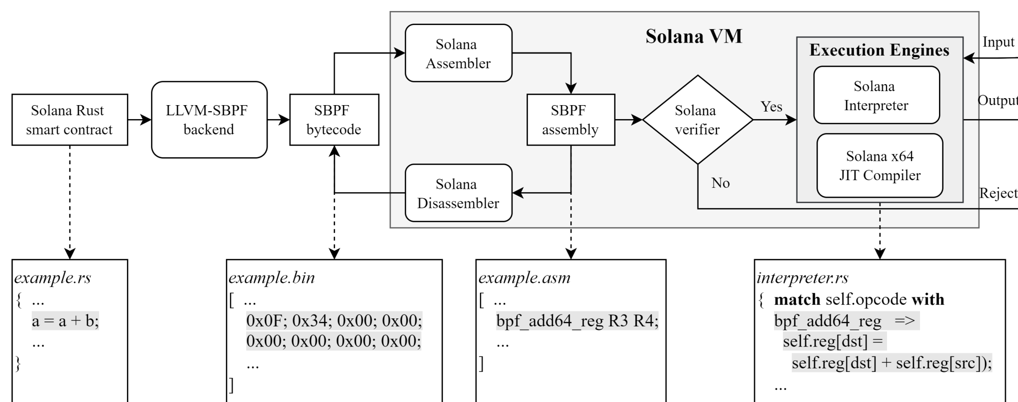

Fig. 2. Solana eBPF VM structure.

As shown in Figure 2, we explain the workflow of the Solana VM by an example. The input smart contract has the code fragment of an addition of two variables a and b of types $u64$, it is compiled into a 64-bit bytecode $[\ldots;\ 0x0F;\ 0x34;\ 0x00;\ 0x00;\ 0x00;\ 0x00;\ 0x00;\ 0x00;\ \ldots]$ where the opcode $0F$ represents the Solana 64-bit addition instruction, the variables a and b are allocated into registers $R3$ and $R4$ respectively, and the rest fields in this 64-bit bytecode are all zero. This bytecode is further translated into the Solana assembly instruction $BPF_ADD64_REG\ R3\ R4$, and executed by the Solana interpreter using a pattern-match statement.

The SBPF instruction set has undergone multiple iterations over time, resulting in the coexistence of versions $V1$ and $V2$. This versioning ensures backward compatibility with previously deployed on-chain eBPF programs while enabling the introduction of new features.

Several key distinctions between the Solana eBPF VM and the Linux eBPF complicate our formalization:

- *Termination*: SBPF uses CU as a metric to gauge resource consumption in Solana's runtime, whereas eBPF employs static analysis techniques via an offline verifier to ensure termination. This runtime CU-based termination adds complexity to SBPF's execution engines (e.g., JIT).
- *Dynamic Stack Frame*: While eBPF has a fixed-size stack (512B), SBPF supports dynamic stack frames, requiring the SBPF VM to manage the calling conventions.
- *Solana ISA*: SBPF inherits most general-purpose instructions from eBPF but excludes certain instructions (e.g., atomic operations) for on-chain transaction safety. Moreover, SBPF introduces 13 new instructions tailored to the blockchain context.
  - *Version Compatibility*: SBPF maintains two different ISAs ($V1$ and $V2$), requiring the Solana VM to handle both versions across all components. This version management introduces additional potential for errors.
  - *New Instructions*: Solana extends its ISA with new features, including signed instructions and 128-bit operand support. SBPF has special instructions to modify the stack pointer while eBPF's stack pointer is always read-only.
  - *Differing Semantics*: SBPF and eBPF have different behaviours for the same opcodes. For example, in SBPF, the 32-bit *add* instruction uses a signed extension, while in eBPF, it uses an unsigned extension. Furthermore, the EXIT instruction in eBPF terminates bytecode execution, whereas, in SBPF, it also serves as a callback mechanism for SBPF function calls.

3 Overview

This section presents an overview of our methodology, which, to the best of our knowledge, is the first and most comprehensive formal semantics of the SBPF ISA. In the subsequent sections, we will demonstrate the faithfulness and the usability of our formal model by applying it to various scenarios.

As illustrated in Figure 3, we begin by formalizing the original rust-implemented SBPF VM, into an abstract high-level semantic model in Isabelle/HOL. This formalization serves as the theoretical foundation, supporting both the executable semantics for validation and the formalization of the core components in Solana VM.

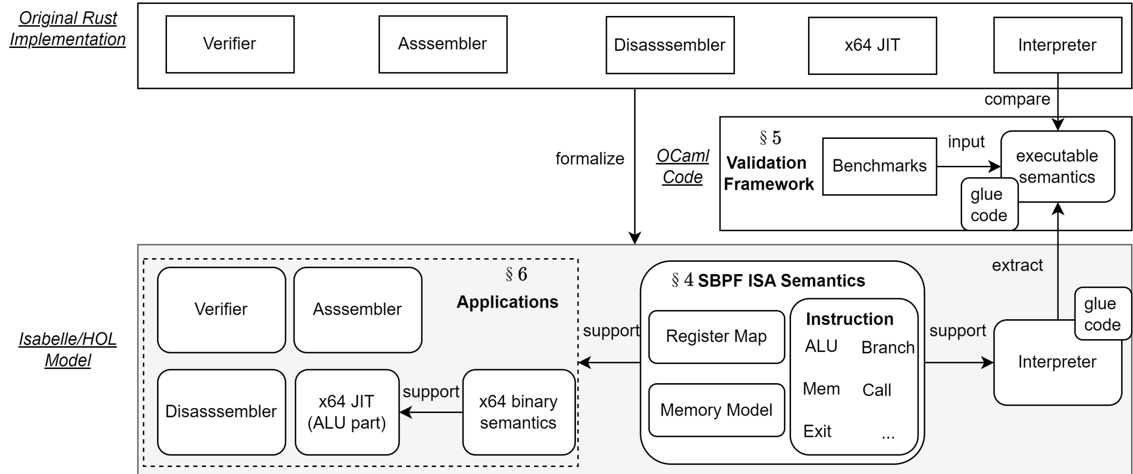

Fig. 3. Overview of the Solana ISA semantics and its applications.

Semantics. (§4) The formal semantics for the SBPF ISA captures the entire instruction set and defines two key components of the execution state: the Solana register map and a general-purpose memory model. This formalization provides a high-level specification for the Solana VM interpreter, ensuring precise modelling for all instruction semantics.

Validation Framework. (§5) Based on the Solana interpreter specification in Isabelle/HOL, we leverage the Isabelle/HOL-to-OCaml extraction mechanism to generate executable semantics. To overcome the extraction limitations, we introduce lightweight glue code at both the Isabelle/HOL and OCaml levels. Our validation framework includes the automatic generation of extensive benchmarks to test the consistency between the extracted executable OCaml code and the original Rust-based Solana interpreter.

Solana Applications. (§6) Our semantics also enables the formalization of several components of the Solana VM, including the verifier, assembler, disassembler, and portions of the x86-64 JIT compiler. Additionally, we provide a binary semantics of the x86-64 model for future verification, covering all x86-64 instructions utilized by the JIT compiler.

4 Formalization of SBPF Semantics

This section presents the formal syntax, program state, and semantics of SBPF, along with the formalization of the Solana interpreter.

4.1 Syntax

The formal syntax of the SBPF ISA is depicted in Figure 4. The SBPF ISA has 10 general-purpose registers, a frame pointer (FP) register, and a program counter (PC). Memory access in SBPF is facilitated through the size of the accessed memory block *mb*. SBPF provides common arithmetic, logic, and signed/unsigned condition operations. In particular, the division and modulo in SBPF have different variants according to the specific ISA version, they are called version-related operations (*vop*) in the paper. The first operator (if exists) of an SBPF instruction is a destination register, and the second operator (*sop*) is usually a source register or a 32-bit immediate number.

$$
\begin{array}{rlrl}
\text{(Register)} & r & ::= & R0 \mid R1 \mid R2 \mid \ldots \mid R9 \mid FP \mid PC \\
\text{(MemBlock)} & mb & ::= & M8 \mid M16 \mid M32 \mid M64 \\
\text{(ALUOp)} & aop & ::= & add \mid sub \mid mul \mid mov \mid or \mid and \mid xor \mid lsh \mid rsh \mid arsh \\
\text{(VersionOp)} & vop & ::= & div \mid mod \\
\text{(Condition)} & cop & ::= & = \mid \neq \mid <_u \mid \leq_u \mid \geq_u \mid >_u \mid <_s \mid \leq_s \mid \geq_s \mid >_s \mid \ldots \\
\text{(SecondOp)} & sop & ::= & r \mid imm \\
\text{(Instruction)} & ins & ::= & \texttt{ALU32}\ aop\ r_d\ sop \mid \texttt{ALU64}\ aop\ r_d\ sop \mid \\
 & & & \texttt{MDM32}\ vop\ r_d\ sop \mid \texttt{MDM64}\ vop\ r_d\ sop \mid \\
 & & & \texttt{BE}\ r_d\ imm \mid \\
 & & (*V1*) & \texttt{NEG32}\ r_d \mid \texttt{NEG64}\ r_d \mid \texttt{LE}\ r_d\ imm \mid \\
 & & (*V2*) & \texttt{PQR32}\ vop\ r_d\ sop \mid \texttt{PQR64}\ vop\ r_d\ sop \mid \\
 & & (*V2*) & \texttt{UHMUL}\ r_d\ sop \mid \texttt{SHMUL}\ r_d\ sop \mid \\
 & & (*V2*) & \texttt{LDDW}\ r_d\ imm\ imm \mid \texttt{HOR64}\ dst\ imm \mid \texttt{ADD_STK}\ imm \mid \\
 & & & \texttt{JA}\ ofs \mid \texttt{JUMP}\ cop\ r_d\ sop\ ofs \mid \\
 & & & \texttt{LD}\ mb\ r_d\ r_s\ ofs \mid \texttt{ST}\ mb\ r_d\ sop\ ofs \mid \\
 & & & \texttt{CALL_REG}\ r_s\ imm \mid \texttt{CALL_IMM}\ r_s\ imm \mid \texttt{EXIT}
\end{array}
$$

Fig. 4. The syntax of the SBPF formal model

According to the version of SBPF ISA, the SBPF instructions are split into:

- *common ISA used in all versions*: the 32-bit and 64-bit ALU instructions, the unsigned division and modulo instructions (MDM), the byte-swap instruction BE, JUMP instructions with a signed 16-bit offset, memory load and store with different memory block sizes, call instructions (with a register value or an immediate number), and the exit instruction.
- *SBPF_v1 specific ISA*: the 32-bit and 64-bit negation instructions (NEG), and a byte-swap instruction LE converting to litter-endian format.
- *SBPF_v2 specific ISA*: the 32-bit and 64-bit signed division and modulo instructions (PQR), 128-bit unsigned (UHMUL) and signed (SHMUL) multiplication, and the load double-words instruction LDDW, high 32-bit bitwise-or instruction HOR64, and the special stack pointer modification instruction ADD_STK.

4.2 Semantics

Program State. The normal program state is a 5-tuple $\mathcal{S} ::= \langle \mathcal{R}, \mathcal{M}, Stack, Version, call_map \rangle$, consisting of

365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416

- the register state $\mathcal{R}$ is a mapping from the Solana registers to 64-bit integers: $\mathcal{R} \in r \rightarrow int64$;
- the memory model $\mathcal{M}$ is a partial mapping from a 64-bit address to a byte: $\mathcal{M} \in int64 \rightharpoonup int8$, and it provides basic memory operations, detailed in subsubsection 4.2.5;
- the stack state $Stack ::= \langle call_depth, stack_pointer, call_frame_list \rangle$ records the current call depth, the stack pointer, and the list of current call frames. Each frame $cf ::= \langle caller_saved_regs,\ frame_pointer,\ return_addr \rangle$ includes the value of the caller save registers (*i.e.,* $R6 - R9$), the caller frame pointer, and the return address;
- The Solana ISA version: $V1$ for the legacy ISA or $V2$ for the current ISA;
- The function call information is a partial mapping from a 32-bit key to a 64-bit value representing the start address of a function: $call_map \in int32 \rightharpoonup int64$.

There are also three specific states: *EFlag* captures a runtime exception message *e.g.,* the value of the source register is zero when interpreting a division instruction, *Err* represents potential undefined behaviours, and *Success* v indicates the normal termination of the Solana VM with the return value v.

Notation. Before formalizing the semantics of Solana instructions, we declare the following notations:

- $\mathcal{S}.X$ represents the X field of the program state $\mathcal{S}$, *e.g.,* $\mathcal{S}.\mathcal{R}$ is the register state of $\mathcal{S}$.
- $\mathcal{S}\{X \leftarrow Y\}$ modifies the X (sub-)field of $\mathcal{S}$ with the value Y. *e.g.,* $\mathcal{S}\{PC \leftarrow v\}$ uses v to update the value of PC register in the register map of the state $\mathcal{S}$.
- $[[r]]_{ty}$ with a type casting suffix ty ($\in \{s32, u32, s64, u64\}$) indicates the signed/unsigned extended value of register r in the state, we overload this notation $[[sop]]_{ty}$: if sop is a register r, returns $[[r]]_{ty}$, otherwise indicates a signed/unsigned extended value of an immediate number. $[[r]]$ is by default $[[r]]_{u64}$ for simplification.
- For option types, $\lfloor v \rfloor$ (*i.e., Some* v) indicates success, and $\emptyset$ (*i.e., None*) indicates failure.
- $\mathcal{S} \xrightarrow{ins} \mathcal{S}'$ represents one-step execution of the SBPF instruction ins, performing a semantics transition from the initial state $\mathcal{S}$ to the final state $\mathcal{S}'$.

4.2.1 Semantics: ALU Instructions. The Solana ALU instructions have complicated behaviours due to the different versions, signed/unsigned semantics, etc. Most ALU instructions in Solana have both 32-bit and 64-bit operators, and all of them have been formalized in Isabelle/HOL. We mainly introduce the transition rules of 32-bit operations in the following because they have much more complex type-casting behaviours.

$$\texttt{eval_aop32}(aop, r_d, sop, \mathcal{R}) \stackrel{\text{def}}{=} \begin{cases} (u64)([[r_d]]_{s32} + [[sop]]_{s32}) & , \textbf{ if } aop = add \\ (u64)([[r_d]]_{u32} \mid [[sop]]_{u32}) & , \textbf{ if } aop = or \\ \dots & \\ (u64)([[r_d]]_{s32} \gg ([[sop]]_{u32} \,\&\, 31)) & , \textbf{ if } aop = arsh \end{cases}$$

For 32-bit ALU instructions, *add*, *sub*, and *mul* adopt explicit signed extension semantics[1], as defined in `eval_aop32`, *arsh* requires the signed/unsigned extension for the first/second operator with a safe masking operation for avoiding shift errors, and the rest perform unsigned extension behaviours. The semantics rule *ALU32-Normal* represents that the normal execution of `ALU32` updates the destination register with the evaluated result, and changes PC to point to the next 64-bit instruction.

[1]https://github.com/solana-labs/solana/issues/32924

417
418
419
420

$$\frac{\texttt{eval_aop32}(aop, r_d, sop, \mathcal{S}.\mathcal{R}) = v}{\mathcal{S} \xrightarrow{\texttt{ALU32}\ aop\ r_d\ sop} \mathcal{S}\{r_d \leftarrow v,\ PC \leftarrow [\![PC]\!] + 1\}}\ (ALU32\text{-}Normal)$$

421 For 32-bit *div* and *mod* instructions,

422
423
424
425
426
427

- *MDM32-Normal*: when the value of the second operator is not zero, the semantics rule performs a normal transition;
- *MDM32-Err*: If the second operator is an immediate number, the transition goes to *Err* when the immediate is 0;
- *MDM32-EFlag*: If the second operator is a register, the state is changed into *EFlag* when the value is 0 at runtime.

428
429
430
431
432
433

$$\frac{(u64)([\![r_d]\!]_{u32}\ vop\ [\![sop]\!]_{u32}) = v \quad [\![sop]\!]_{u32} \neq 0}{\mathcal{S} \xrightarrow{\texttt{MDM32}\ vop\ r_d\ sop} \mathcal{S}\{r_d \leftarrow v,\ PC \leftarrow [\![PC]\!] + 1\}}\ (MDM32\text{-}Normal)$$

434
435
436

$$\frac{imm = 0}{\mathcal{S} \xrightarrow{\texttt{MDM32}\ vop\ r_d\ imm} Err}\ (MDM32\text{-}Err) \qquad \frac{[\![r_s]\!]_{u32} = 0}{\mathcal{S} \xrightarrow{\texttt{MDM32}\ vop\ r_d\ r_s} EFlag}\ (MDM32\text{-}EFlag)$$

437
438
439

The negation instructions only exist in the legacy Solana_v1 ISA. If the ISA version is not $V1$, the transition goes to *Err* state (*NEG32-Err*). Otherwise, a negation operation with two kinds of type casting is performed (*NEG32-Normal*).

440
441
442

$$\frac{\mathcal{S}.Version \neq V1}{\mathcal{S} \xrightarrow{\texttt{NEG32}\ r_d} Err}\ (NEG32\text{-}Err) \qquad \frac{\mathcal{S}.Version = V1 \quad (u64)(-[\![r_d]\!]_{i32}) = v}{\mathcal{S} \xrightarrow{\texttt{NEG32}\ r_d} \mathcal{S}\{r_d \leftarrow v,\ PC \leftarrow [\![PC]\!] + 1\}}\ (NEG32\text{-}Normal)$$

443 In the following semantics rules, we omit the *Err*/*EFlag*-related state transitions for simplification.

444
445
446
447
448
449
450

4.2.2 Semantics: Byte-swap Instructions. We first declare the atomic function $\texttt{byte}(v, n)$ which gets the nth-byte of value v, *e.g.*, $\texttt{byte}(0x1234, 0) = 0x34$ and $\texttt{byte}(0x1234, 1) = 0x12$. Then the byte-swap functions $\texttt{to_be}(v, sz)$ and $\texttt{to_le}(v, sz)$, accepting unsigned sz-bytes value v and returning the value with the same size, are defined as follows, where res and v satisfy the relation $\forall n.\ n \leq sz \ \rightarrow \texttt{byte}(res, n) = \texttt{byte}(v, sz - n)$.

451
452
453

$$\texttt{to_be}(v, sz) \stackrel{\text{def}}{=} \begin{cases} v & , \textbf{if}\text{ target is big-endian} \\ res & , \textbf{if}\text{ target is litten-endian} \end{cases} \quad \texttt{to_le}(v, sz) \stackrel{\text{def}}{=} \begin{cases} res & , \textbf{if}\text{ target is big-endian} \\ v & , \textbf{if}\text{ target is litten-endian} \end{cases}$$

454
455 In this paper, we mainly discuss litten-endian architectures *e.g.*, x86 and x86-64.

456
457

SBPF has an instruction *BE* for all versions. The normal transition rule *BE-Normal* converts an unsigned *imm*-bit integer to a big-endian using the function *to_be*, where *imm* is limited to 16, 32, or 64.

458
459
460
461

$$\frac{imm \in \{16,\ 32,\ 64\} \quad (u64)(\texttt{to_be}([\![r_d]\!]_{uimm},\ imm/8 - 1)) = v}{\mathcal{S} \xrightarrow{\texttt{BE}\ r_d\ imm} \mathcal{S}\{r_d \leftarrow v,\ PC \leftarrow [\![PC]\!] + 1\}}\ (BE\text{-}Normal)$$

462
463

The legacy Solana ISA also includes a specific instruction: *LE*. Similarly, the normal transition rule *LE-Normal* converts an unsigned *imm*-bit integer to a little-endian, but limits the *LE* instruction specific to the Solana_v1 version.

464
465
466
467

$$\frac{imm \in \{16,\ 32,\ 64\} \quad (u64)(\texttt{to_le}([\![r_d]\!]_{uimm},\ imm/8 - 1)) = v \quad \mathcal{S}.Version = V1}{\mathcal{S} \xrightarrow{\texttt{LE}\ r_d\ imm} \mathcal{S}\{r_d \leftarrow v,\ PC \leftarrow [\![PC]\!] + 1\}}\ (LE\text{-}Normal)$$

468

469 470 471 472

4.2.3 Semantics: Solana_v2 specific Instructions. The Solana_v2 ISA provides a set of explicitly signed operations for *div* and *mod*, named *PQR*, and the 128-bit multiplication. We mainly show the signed semantics rules for normal transitions (*PQR32-Normal* and *SHMUL-Normal*).

$$\frac{(u64)([\![r_d]\!]_{i32}\ vop\ [\![sop]\!]_{i32}) = v \quad [\![sop]\!]_{i32} \neq 0 \quad \mathcal{S}.Version \neq V1}{\mathcal{S} \xrightarrow{\texttt{PQR32}\ vop\ r_d\ sop} \mathcal{S}\{r_d \leftarrow v,\ PC \leftarrow [\![PC]\!] + 1\}}\ (PQR32\text{-}Normal)$$

$$\frac{(u64)(([\![r_d]\!]_{i128} \times [\![sop]\!]_{i128}) \gg 64) = v \quad \mathcal{S}.Version \neq V1}{\mathcal{S} \xrightarrow{\texttt{SHMUL}\ r_d\ sop} \mathcal{S}\{r_d \leftarrow v,\ PC \leftarrow [\![PC]\!] + 1\}}\ (SHMUL\text{-}Normal)$$

There are three specific instructions in the Solana_v2 ISA:

- *LDDW*: Loads a 64-bit integer (usually a memory address) into the destination register where the integer is split into the low 32-bit integer stored in the immediate field and the high 32-bit one stored in the next 64-bit binary. The *LDDW* instruction has a 128-bit size, therefore the value of *PC* is increased by 2.

$$\frac{[\![imm_l]\!]_{u64} \mid ([\![imm_h]\!]_{u64} \ll 32) = v \quad \mathcal{S}.Version \neq V1}{\mathcal{S} \xrightarrow{\texttt{LDDW}\ r_d\ imm_l\ imm_h} \mathcal{S}\{r_d \leftarrow v,\ PC \leftarrow [\![PC]\!] + 2\}}\ (LDDW\text{-}Normal)$$

- *HOR64*: Modifies the high 32-bit of the destination register using the bitwise OR operation.

$$\frac{[\![r_d]\!]_{u64} \mid ([\![imm]\!]_{u64} \ll 32) = v \quad \mathcal{S}.Version \neq V1}{\mathcal{S} \xrightarrow{\texttt{HOR64}\ r_d\ imm} \mathcal{S}\{r_d \leftarrow v,\ PC \leftarrow [\![PC]\!] + 1\}}\ (HOR64\text{-}Normal)$$

- *ADD_STK*: Modifies the stack pointer of the Solana VM.

$$\frac{\mathcal{S}.Version \neq V1}{\mathcal{S} \xrightarrow{\texttt{ADD_STK}\ imm} \mathcal{S}\{stack_pointer \leftarrow stack_pointer + [\![imm]\!]_{u64},\ PC \leftarrow [\![PC]\!] + 1\}}\ (ADD_STK\text{-}Normal)$$

4.2.4 Semantics: Jump Instructions. The evaluation function `eval_cond` is simply defined as follows:

$$\texttt{eval_cond}(cop, v0, v1) \stackrel{\text{def}}{=} \begin{cases} v0 = v1 & ,\ \textbf{if}\ cop\ is\ = \\ v0 \neq v1 & ,\ \textbf{if}\ cop\ is\ \neq \\ \ldots \end{cases}$$

The jump instructions have the semantics: the target *PC* could be either the next instruction (*Jump-F*) or the offset computation $[\![PC]\!] + ofs + 1$ (*Jump-T*), relying on whether the value of the destination register and the second operator satisfy the condition *cop* or not. The jump-always (JA) instruction always performs the offset computation (*JA-Normal*).

$$\frac{}{\mathcal{S} \xrightarrow{\texttt{JA}\ ofs} \mathcal{S}\{PC \leftarrow [\![PC]\!] + ofs + 1\}}\ (JA\text{-}Normal)$$

$$\frac{\texttt{eval_cond}(cop, [\![r_d]\!], [\![sop]\!]) = True}{\mathcal{S} \xrightarrow{\texttt{JUMP}\ cop\ r_d\ sop\ ofs} \mathcal{S}\{PC \leftarrow [\![PC]\!] + ofs + 1\}}\ (Jump\text{-}T) \quad \frac{\texttt{eval_cond}(cop, [\![r_d]\!], [\![sop]\!]) = False}{\mathcal{S} \xrightarrow{\texttt{JUMP}\ cop\ r_d\ sop\ ofs} \mathcal{S}\{PC \leftarrow [\![PC]\!] + 1\}}\ (Jump\text{-}F)$$

4.2.5 Semantics: Memory Instructions. Our memory model, adopting the little-endian style, provides some basic operations *e.g.,*

- $\texttt{load}(mb,\ \mathcal{M},\ addr) = \lfloor v \rfloor$: Reads *mb*-byte value v at starting address *addr* from $\mathcal{M}$,

521
522
523
524
525
526
527
528

$$\texttt{load}(mb,\ \mathcal{M},\ addr) \stackrel{\text{def}}{=} \begin{cases} \lfloor v0 \rfloor & \text{, } \textbf{if } mb = M8 \wedge \lfloor \texttt{byte}(v1, 0) \rfloor = \mathcal{M}\ addr \\ \lfloor v1 \rfloor & \text{, } \textbf{if } mb = M16 \wedge (\forall i.\ 0 \leq i \leq 1 \longrightarrow \lfloor \texttt{byte}(v1, i) \rfloor = \mathcal{M}\ (addr + i)) \\ \lfloor v3 \rfloor & \text{, } \textbf{if } mb = M32 \wedge (\forall i.\ 0 \leq i \leq 3 \longrightarrow \lfloor \texttt{byte}(v1, i) \rfloor = \mathcal{M}\ (addr + i)) \\ \lfloor v7 \rfloor & \text{, } \textbf{if } mb = M64 \wedge (\forall i.\ 0 \leq i \leq 7 \longrightarrow \lfloor \texttt{byte}(v1, i) \rfloor = \mathcal{M}\ (addr + i)) \\ \emptyset & \text{, } \textbf{Otherwise} \end{cases}$$

529
530
531

- $\texttt{store}(mb,\ \mathcal{M},\ addr,\ v) = \lfloor \mathcal{M}' \rfloor$: Writes mb-byte value v into $\mathcal{M}$' at starting address $addr$, and returns the modified memory $\mathcal{M}'$. We write $\mathcal{M}\{loc \mapsto b\}$ for updating the cell of the address loc in $\mathcal{M}$ with a byte value b.

532
533
534
535
536
537
538
539
540

$$\texttt{store}(mb,\ \mathcal{M},\ addr,\ v) \stackrel{\text{def}}{=} \begin{cases} \lfloor \mathcal{M}\{addr \mapsto \texttt{byte}(v, 0)\} \rfloor & \text{, } \textbf{if } mb = M8 \\ \lfloor \mathcal{M}\{addr + i \mapsto \texttt{byte}(v, i)\} \rfloor & \text{, } \textbf{if } mb = M16 \wedge 0 \leq i \leq 1 \\ \lfloor \mathcal{M}\{addr + i \mapsto \texttt{byte}(v, i)\} \rfloor & \text{, } \textbf{if } mb = M32 \wedge 0 \leq i \leq 3 \\ \lfloor \mathcal{M}\{addr + i \mapsto \texttt{byte}(v, i)\} \rfloor & \text{, } \textbf{if } mb = M64 \wedge 0 \leq i \leq 7 \\ \emptyset & \text{, } \textbf{Otherwise} \end{cases}$$

541
542

The memory instructions use those operations to perform semantics transitions which update either the destination register (LD) or the memory (ST).

543
544
545
546
547

$$\frac{\texttt{load}(mb,\ \mathcal{S}.\mathcal{M},\ [[r_s]] + ofs) = \lfloor v \rfloor}{\mathcal{S} \xrightarrow{\texttt{LD}\ mb\ r_d\ r_s\ ofs} \mathcal{S}\{r_d \leftarrow v,\ PC \leftarrow [[PC]] + 1\}}\ (Load) \qquad \frac{store(mb,\ \mathcal{S}.\mathcal{M},\ [[r_d]] + ofs,\ [[sop]]) = \lfloor \mathcal{M}' \rfloor}{\mathcal{S} \xrightarrow{\texttt{ST}\ mb\ r_d\ sop\ ofs} \mathcal{S}\{\mathcal{M} \leftarrow \mathcal{M}',\ PC \leftarrow [[PC]] + 1\}}\ (\text{Store})$$

548
549
550
551
552

4.2.6 Semantics: Call Instructions. SBPF provides two call instructions: call with register (Call_REG) and call with immediate (Call_IMM). These instructions share similar semantics: first, computing the target pc, then pushing the current frame onto the global frame list, followed by updating the register map. The primary distinction between the two lies in the computation of the target PC.

553
554

For Call_REG, the target PC is evaluated using:

555
556
557

$$\texttt{eval_target_pc_reg}(\mathcal{S}, r_s, imm) \stackrel{\text{def}}{=} \begin{cases} [[R_{imm}]] & \text{, } \textbf{if } \mathcal{S}.Version = V1 \\ [[r_s]] & \text{, } \textbf{if } \mathcal{S}.Version \neq V1 \end{cases}$$

558
559
560
561

Where, in Solana_v1, the register index is determined by the immediate value imm, and for other Solana versions, the target PC is stored in the source register r_s. Once the target PC is determined, the call instruction invokes the function push_frame, which performs the following steps:

562
563
564
565
566
567
568
569
570
571

$$\begin{aligned} &\texttt{push_frame}(\mathcal{S}) \stackrel{\text{def}}{=} \\ &\quad \textbf{let } nfr\ = <\ [\ [[R6]], [[R7]], [[R8]], [[R9]]\ ],\ [[FP]],\ [[PC]] + 1 > \textbf{in} \\ &\quad \textbf{let } nsp\ =\ \textbf{if } \mathcal{S}.Version\ =\ V1\ \textbf{then } \mathcal{S}.stack_pointer +\ stack_frame_size\ \textbf{else } \mathcal{S}.stack_pointer\ \textbf{in} \\ &\quad \textbf{let } stk\ = < \mathcal{S}.call_depth + 1,\ nsp,\ \mathcal{S}.call_frame_list\{\mathcal{S}.call_depth \mapsto nfr\} >\ \textbf{in} \\ &\qquad (stk,\ nsp) \end{aligned}$$

572

- Creates a new frame nfr, storing the values of caller-saved registers ($R6$ to $R9$), the FP register, and the return address ($[\![\mathrm{PC}]\!] + 1$).
- Optionally adjusts the stack pointer based on the stack frame size (default: 4096, or 8192 if specified), as Solana_v1 uses a dynamic stack frame size.
- Pushes this frame onto the call stack (*i.e.,* stored at position $call_depth$), increments the call depth by 1, and updates the stack pointer.

$$\frac{imm \in [0,9] \quad \texttt{eval_target_pc_reg}(\mathcal{S}, r_s, imm) = v \quad \texttt{push_frame}(\mathcal{S}) = (stk, nsp)}{\mathcal{S} \xrightarrow{\texttt{Call_REG}\ r_s\ imm} \mathcal{S}\{PC \leftarrow v,\ FP \leftarrow nsp,\ Stack \leftarrow stk\}}\ (Call_REG)$$

After updating the call stack, `Call_REG` modifies the PC register to point to the target PC, updates the FP register with the new stack pointer, and updates the execution state with the modified stack.

The `Call_IMM` instruction in Solana operates in two modes, depending on the index of the source register r_s (whether s is zero or not):

- **External Call:** Invokes system APIs provided by the Solana platform, relying on Rust's calling convention, which is analogous to the Linux eBPF call mechanism.
- **Internal Call:** Executes functions defined within the bytecode of the Solana program.

Both modes utilize a partial function call map to compute the target address, represented by:

$$\texttt{eval_target_pc_imm}(\mathcal{S}, imm) \stackrel{\text{def}}{=} \mathcal{S}.call_map(imm)$$

In this paper, we focus primarily on formalizing the internal call mechanism, which operates similarly to `Call_REG`. External calls are trusted and abstracted out of the formalization.

$$\frac{r_s \neq R0 \quad \texttt{eval_target_pc_imm}(\mathcal{S},\ imm) = \lfloor v \rfloor \quad \texttt{push_frame}(\mathcal{S}) = (stk, nsp)}{\mathcal{S} \xrightarrow{\texttt{Call_IMM}\ r_s\ imm} \mathcal{S}\{PC \leftarrow v,\ FP \leftarrow nsp,\ Stack \leftarrow stk\}}\ (Call_IMM)$$

4.2.7 Semantics: Exit Instruction. The transition moves to the *Success* state when the call depth is 0, indicating that all SBPF calls have returned (*EXIT-Normal*). If the call depth is greater than 0, the transition performs a callback by removing the current frame from the stack using the `pop_frame` function defined as follows (*Exit-Call*).

$$\begin{aligned}
&\texttt{pop_frame}(\mathcal{S}) \stackrel{\text{def}}{=}\\
&\quad \textbf{let}\ (csl, osp, ra)\ =\ \mathcal{S}.call_frame_list[\mathcal{S}.call_depth - 1]\ \textbf{in}\\
&\quad \textbf{let}\ nsp\ =\ \textbf{if}\ \mathcal{S}.Version = V1\ \textbf{then}\ osp - stack_frame_size\ \textbf{else}\ osp\ \textbf{in}\\
&\quad \textbf{let}\ stk\ = < \mathcal{S}.call_depth - 1,\ nsp,\ \mathcal{S}.call_frame_list >\ \textbf{in}\\
&\qquad \mathcal{S}\{R6 \leftarrow csl[0], R7 \leftarrow csl[1], R8 \leftarrow csl[2], R9 \leftarrow csl[3], FP \leftarrow nsp, PC \leftarrow ra, Stack \leftarrow stk\}
\end{aligned}$$

- **Stack:** Removes the top frame tf from the call stack, decrementing the call depth by 1. If the Solana ISA version supports dynamic stack frames, the stack pointer is updated accordingly.
- **Registers:** Restores the caller-saved registers $R6$ to $R9$, the frame pointer FP, and the program counter PC from the corresponding fields of the top frame tf.

$$\frac{\mathcal{S}.call_depth = 0}{\mathcal{S} \xrightarrow{\texttt{Exit}} Success\ [[R0]]}\ (Exit\text{-}Normal) \qquad \frac{\mathcal{S}.call_depth > 0 \quad \texttt{pop_frame}(\mathcal{S}) = \mathcal{S}'}{\mathcal{S} \xrightarrow{\texttt{Exit}} \mathcal{S}'}\ (Exit\text{-}Call)$$

625 *Interpreter.* All semantics rules are integrated as a semantics function in Isabelle/HOL, *i.e.,* '**step** : $ins \Rightarrow \mathcal{S} \Rightarrow \mathcal{S}$'. We
626 formalize the Solana interpreter as the function '**interpreter** : $nat \Rightarrow byte\ list \Rightarrow \mathcal{S} \Rightarrow \mathcal{S}$' to execute the input smart
627 contract bytecode (*i.e.,* a list of bytes) by invoking *step*, where the first parameter is a fuel for VM termination.
628
629
630 ## 5 Validation of Semantics

631 We have manually formalized the complete SBPF instruction set (excluding external calls) in Isabelle/HOL, based on
632 the original interpreter implementation in Rust. Given the inherent complexity of the SBPF ISA and the potential
633 nuances of Rust semantics, as discussed in subsection 1.1, there are reasonable concerns about whether our formal
634 model accurately reflects the behaviour of the original implementation.
635
636 To address these concerns and build confidence in the accuracy of our formalization, this section outlines the
637 validation process of our formal semantics. This validation is achieved through an executable version generated in
638 OCaml, obtained via the Isabelle/HOL extraction mechanism.
639
640
641 ### 5.1 Validation Framework

642 We have developed a test framework to validate the executable semantics of our Isabelle/HOL model. The primary
643 challenge we encountered is that the extracted OCaml code is not easily human-readable and difficult to work with.
644
645
646 *Problems.* We utilize the Isabelle/HOL extraction mechanism to generate the executable semantics in OCaml. However,
647 our source Isabelle/HOL model relies on the "Word" library to formalize the signed and unsigned semantics of SBPF, which
648 poses considerable difficulty for efficient extraction. As a result, the Isabelle/HOL extraction translates '**interpreter** :
649 $nat \Rightarrow byte\ list \Rightarrow \mathcal{S} \Rightarrow \mathcal{S}$' into OCaml code that follows a cumbersome and less intuitive style (we call it as a
650 constructive style). For instance, 'byte' is a type synonym of '8 word' in Isabelle/HOL, and its extracted representation
651 in OCaml is 'num1 bit0 bit0 bit0 word'.
652
653

```
val interpreter : nat -> num1 bit0 bit0 bit0 word list -> bpf_state -> bpf_state
type num = One | Bit0 of num | Bit1 of num;;
type int = Zero_int | Pos of num | Neg of num;;
type 'a word = Word of int;;
type nat = Zero_nat | Suc of nat;;
```

654
655
656
657
658
659 Although Isabelle/HOL can extract some types, such as 'bool' and 'list', into native OCaml types, its code generator
660 struggles to map more complex types like 'int', 'word', and 'nat' to their corresponding OCaml native types via **code-**
661 **printing** declarations [Dawson 2009]. This limitation complicates the testing of the executable semantics for the SBPF
662 ISA, as it frequently relies on these types.
663
664 To tackle this limitation, existing work [Lochbihler 2018] introduces the "Native_Word" library, which links formal-
665 ized words in Isabelle/HOL to machine words in target languages (e.g., OCaml). However, our current Isabelle/HOL
666 implementation is built on the "Word" library, and all proofs depend on its lemmas. Adopting the "Native_Word" library
667 would require replacing all existing "Word" definitions, leading to additional proof effort: either re-proving all SBPF
668 properties with "Native_Word" lemmas or proving equivalence between the original "Word" model and using the
669 modified "Native_Word" model for code extraction.
670
671
672
673 *Solutions.* We propose a lightweight and non-invasive approach to relax the limitations of Isabelle/HOL extraction
674 by introducing adaptations that glue native OCaml types with the types extracted from Isabelle/HOL. To minimize
675 changes, we divide the glue code into two layers: Isabelle/HOL-level and OCaml-level glue code.
676

677
678
679
680
681
682
683
684
685
686

- **Isabelle/HOL-level glue code**: The original extracted OCaml code, makes it highly difficult to validate semantics with the constructive input type. Consequently, a new function '**interpreter_test** : $int \Rightarrow int\ list \Rightarrow \ \ldots$', is introduced in Isabelle/HOL to internally invoke the existing '**interpreter**' function to perform computations. This glue code also handles type casting between 'int' and other types, such as 'nat' and 'word', by using pre-defined functions in Isabelle/HOL (e.g., '**nat**' translates an integer into a natural number, '**of_int**' to convert a Isabelle/HOL 'int' into a fixed-size word, and 'map' translates '*int list*' to '*u8 list*'). Users must ensure that the type casting is valid, such as always passing a positive integer as the fuel parameter of '**interpreter_test**'. Since this glue code is implemented in Isabelle/HOL, it can be directly extracted into OCaml.

687
688
689
690
691
692

```
fun interpreter :: "nat => u8 list => bpf_state => bpf_state" where ...

definition interpreter_test :: "int => int list => ..." where ...
interpreter_test fuel prog ... =
    interpreter (nat fuel) (map (λ i. of_int i) prog) ...
```

693
694
695
696
697

- **OCaml-level glue code**: The '**interpreter_test**' function provides an interface for testing that only uses 'int'-related types (named *hol_int* to avoid ambiguity). Consequently, the OCaml-level glue code '*int_of_standard_int*' focuses on translating between the generated 'int' types from Isabelle/HOL and the native 'int64' type from the OCaml standard library, and '*interpreter_test_ocaml*' provides a user-friendly interface for OCaml testing.

698
699
700
701
702
703
704
705

```
type hol_int = Zero_int | Pos of num | Neg of num;;
val interpreter_test : hol_int -> hol_int list -> ...
val int_of_standard_int : int64 -> hol_int
val interpreter_test_ocaml : int64 -> int64 list -> ...
let interpreter_test_ocaml fuel prog ... =
  interpreter_test (int_of_standard_int fuel) (List.map int_of_standard_int prog) ...
```

706
707
708
709
710
711
712
713

Framework. Our validation framework, illustrated in Figure 5, operates as follows: Given an input test case, the original Rust implementation of the Solana interpreter, named '*interpreter_rust*', produces an output, *Output*1. Simultaneously, the extracted OCaml function '*interpreter_test_ocaml*', along with the glue code, executes the same test case and produces a second output, *Output*2. If the two results match, the test is valid, demonstrating consistency between the original interpreter and the formal semantics. If the results differ, the framework identifies an inconsistency between the low-level Rust code and the high-level Isabelle/HOL model.

714
715

5.2 Validation Benchmarks

716
717
718

We conducted two types of benchmarks to validate the semantics: micro-benchmarks at the instruction level and macro-benchmarks at the program level.

719
720
721
722

- *Micro-benchmarks*: validation of single instructions with randomly generated data to assess the one-step execution of individual SBPF instructions;
- *Macro-benchmarks*: validation of Solana bytecode programs using the official Solana benchmark suite.

723

Instruction-level Validation. As shown in Figure 6, the instruction-level validation follows three steps:

724
725
726
727
728

- A random SBPF instruction *ins* is generated;
- *ins* is executed by both the original *step* function in Rust and the OCaml function **step_test** (the extracted code from **step**, mentioned in Section 4, with glue code);

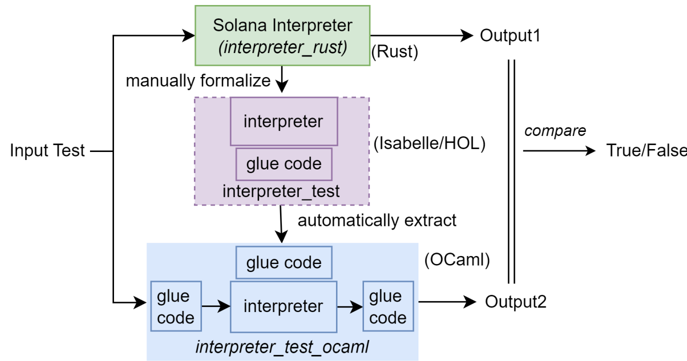

Fig. 5. Validation Framework.

- The results of *step* and **step_test** are compared to ensure consistency.

To generate random SBPF instructions, consider the procedure for generating a random 32-bit move instruction. First, we generate a random index for the destination register, and if the second operand is a register, another index is generated for the source register; otherwise, a 32-bit immediate value is produced. These are then composed into an assembly instruction, which is compiled into binary format by the Solana assembler. We also generate a random register map for testing, where registers $R0 - R9$ hold random 64-bit values, and FP points to the default stack address.

A valid result is achieved under two conditions: either both '*step_rust*' and '*step_test_ocaml*' fail execution, or both functions successfully execute and produce identical values in the destination register.

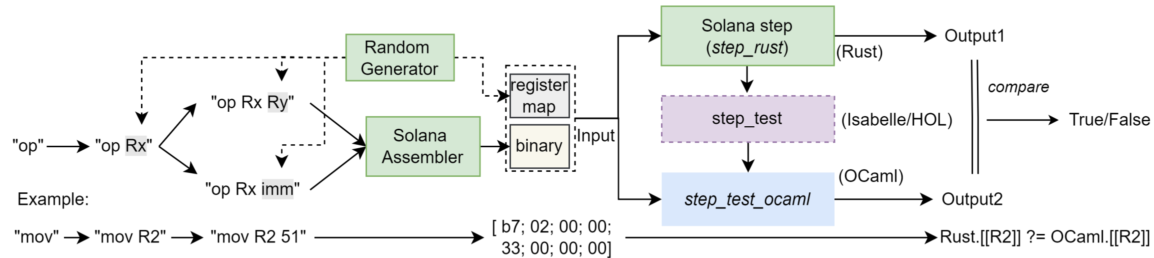

Fig. 6. Instruction-level Validation Framework.

Our instruction-level validation covers ALU, byte-swap, memory load, memory store, and branch instructions. For memory store instructions, which have side effects on memory, comparing entire memory models is time-consuming. Therefore, we generate corresponding memory load instructions to read the modified memory cells and compare the results in the destination register. We exclude `CALL` and `EXIT` instructions, as they require a Solana program structure.

Program-level Validation. To validate the combination of instructions within real-world Solana programs, we utilize the official Solana test suite as input to our validation framework, illustrated in Figure 5. We select 146 out of 160 tests, covering 114 SBPF normal cases and 32 exception cases. The validation proceeds as follows:

- *Normal tests*: each test completes successfully in Rust, and our formal model should reach the *Success* state, with both implementations producing identical results.

- *Exception tests*: each test terminates with an error flag, and our formal model should correspondingly end in the *EFlag* state.

We have to exclude 14 cases that involve external calls, as they require modelling Solana's built-in system calls, which are not yet supported in our formal semantics.

5.3 Validation Results

As anticipated, the validation framework proved instrumental in refining our formal SBPF semantics. Throughout the validation process, we identified some minor issues and some deep potential problems in the initial version of our formalization, all of which were addressed in the final version.

We first employed the official Solana test suite (macro-benchmark) for validation, as it required minimal additional preparation. In certain benchmarks, discrepancies arose between the output of the Solana interpreter (Rust) and our formal model, largely due to minor implementation errors in the formalization. These errors primarily stemmed from irregularities in unsigned or signed type castings within the SBPF interpreter. For example, the signed 128-bit multiplication instruction incorrectly applied unsigned casting $[[R_s]]_{u128}$ in our formalization, when the correct approach required signed casting $[[R_s]]_{i128}$.

```
ebpf::SHMUL64_REG ... => self.reg[dst] =
                    (self.reg[dst] ... as i128) // signed type casting
      .wrapping_mul(self.reg[src] ... as i128) // signed type casting
      .wrapping_shr(64) as u64,
```

Subsequently, we developed micro-benchmarks and an instruction-level validation framework (see Figure 6). This framework successfully uncovered additional inconsistencies, arising from fundamental semantic differences between Rust and Isabelle/HOL. To simplify our explanation, we focus on the modulo (remainder) operator.

Table 1 illustrates four examples of modulo operations in three cases: Rust with signed 16-bit integers, Isabelle/HOL with the integer library, and Isabelle/HOL with the "Word" library.

Language	Rust	Isabelle/HOL int	Isabelle/HOL Word
Example1	(-11 as i16) % (10 as i16) = -1	(-(11::int)) mod (10::int) = 9	(-(11::i16)) mod (10::i16) = 5
Example2	(-19 as i16) % (10 as i16) = -9	(-(19::int)) mod (10::int) = 1	(-(19::i16)) mod (10::i16) = 7
Example3	(11 as i16) % (-10 as i16) = 1	(11::int) mod (-(10::int)) = -9	(11::i16) mod (-(10::i16)) = 11
Example4	(19 as i16) % (-10 as i16) = 9	(19::int) mod (-(10::int)) = -1	(19::i16) mod (-(10::i16)) = 19

Table 1. Semantics-level differences between Rust and Isabelle/HOL: modulo examples.

In Rust, the modulo operator (%) returns the remainder of a division, with the result sharing the sign of the dividend. This is because Rust uses *truncated division*, *i.e.,* the result of the division is truncated toward zero.

In Isabelle/HOL, the modulo operator `mod` is generic across types. For integers, it follows *floor division* semantics, where the division result is rounded toward negative infinity, and the modulo result takes the sign of the divisor, irrespective of the dividend's sign. For instance, `(-(11::int)) mod (10::int)` returns 9, because `(-(11::int)) div (10::int)` yields `-2`, and `-2 - (-11)` is `9`.

The Isabelle/HOL "Word" library's modulo behaviour is less straightforward and initially produced results inconsistent with the Rust implementation. To resolve this, we defined a new modulo (and division) function of word types in Isabelle/HOL to correctly implement *truncated division* semantics, ensuring alignment with Rust's behaviour.

833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884

6 Applications

In this section, we illustrate a few applications of our formal model for Solana. Our goal here is to demonstrate how our model can be extended to formalize other key components of the SBPF ecosystem and prove essential properties. For this reason, the applications provided here are intended as examples, with many formal details omitted, since the goal of this paper is the formalization of SBPF semantics.

The remainder of this section outlines how our formal model can serve as a foundation for:

- *Assembler and Disassembler*: reusing the syntax of our model to formalize both components and proving the consistency property of the assembler-disassembler pair;
- *Verifier*: leveraging the syntax to formalize the Solana verifier, and proving the safety properties of our SBPF semantics based on this formalized verifier;
- *JIT Compiler*: reusing the formal semantics to verify the correctness of individual JIT compilation rules for ALU operations.

6.1 Consistency of Solana Assembler and Disassembler

The SBPF VM has two components: Assembler translates machine-readable SBPF bytecode into human-readable SBPF assembly syntax, and Disassembler performs the opposite direction. We first formalize this pair in Isabelle/HOL based on our SBPF syntax: the formal model of Assembler is constructed by the discussion of each SBPF instruction, and the formal Disassembler is constructed by nested if-structures to analyse all fields of the SBPF bytecode.

We then prove the consistency property which confirms the correctness of this bidirectional translation, we first discuss two lemmas:

- *Assembler implies Disassembler*(Lemma 6.1): given that a list of SBPF assembly instructions is encoded into binary form, prove that the encoded instructions can always be decoded back to the original assembly code.
- *Disassembler implies Assembler*(Lemma 6.2): given that a list of SBPF binary instructions is decoded into assembly form, prove that the decoded instructions can always be re-encoded to reproduce the original binary code.

LEMMA 6.1 (ASSEMBLER_IMPLIES_DISASSEMBLER). *If* **assembler** $l_asm = \lfloor l_bin \rfloor$, *then* **disassembler** $l_bin = \lfloor l_asm \rfloor$

PROOF. It uses proof by induction over the assembly instruction list l_asm: the basic case (l_asm is an empty list) is trivial, another inductive case (l_asm is constructed by the head h and the rest list tl) requires case analysis on each input SBPF instruction and uses the inductive hypothesis to complete the proof. □

LEMMA 6.2 (DISASSEMBLER_IMPLIES_ASSEMBLER). *If* **disassembler** $l_bin = \lfloor l_asm \rfloor$, *then* **assembler** $l_asm = \lfloor l_bin \rfloor$

PROOF. The proof begins by induction over the binary instruction list l_bin, then case analysis on the nested if-structures, which rely on the high degree of proof automation of Isabelle/HOL to solve all subgoals. □

THEOREM 6.3 (CONSISTENCY). **assembler** $l_asm = \lfloor l_bin \rfloor \Longleftrightarrow$ **disassembler** $l_bin = \lfloor l_asm \rfloor$

PROOF. The proof is established directly by applying Lemma 6.1 and Lemma 6.2. □

6.2 Verification of Solana Verifier

The Solana verifier provides basic static checking of input bytecode, *e.g.,* detecting the division-by-zero case when the instruction is the division with an immediate number. Two notations are explained firstly: $L(pc)$ represents the

885 64-binary bytecode of the input program L at location $(pc * 8)$, and $L[pc]$ represents the decoded assembly instruction
886 of the bytecode. We also write $L(pc).X$ to access the X field of the bytecode. Then the formalization of the verifier is
887 based on our formal model mentioned in Section 4, and the whole Solana verifier is split into three checking rules:
888
889
890
$$\mathbf{verifier}(L, sv) \stackrel{\text{def}}{=} \forall\, pc, verifier_comm(L, pc) \wedge verifier_reg(L, pc) \wedge verifier_ins(L, pc, sv)$$
891
892
893 *Common Rule.* stipulates the bytecode list of the input Solana program is i/ Not empty; ii/ The length of the list is an
894 integer multiple of the size of each instruction, *i.e.,* an 8-byte Solana bytecode; iii/ Each instruction has a valid opcode.
895
896
897
$$verifier_comm(L, pc) \stackrel{\text{def}}{=} Length(L) \neq 0 \;\wedge\; Length(L)\%8 = 0 \;\wedge\; L[pc] \in ins$$
898
899 *Register Rule.* stipulates for each instruction, the source register could be within the range of $[0,\ 10]$ in most cases, it
900 couldn't be 10 for the call with register instruction. In contrast, the destination register should be within $[0,\ 9]$ in most
901 cases. The only exception is the store instruction, in this case, the destination register could be $R10$ (*i.e., FP*) because
902 SBPF allows storing a value into the VM stack frame.
903
904
905
906
$$verifier_reg(L, pc) \stackrel{\text{def}}{=} \bigwedge \begin{cases} (0 \leq L(pc).src < 10 \;\vee\; (L(pc).src = 10 \wedge L[pc] \neq \texttt{CALL_REG}\ _\ _)) \\ (0 \leq L(pc).dst < 10 \;\vee\; (L(pc).dst = 10 \wedge L[pc] = \texttt{ST}\ _\ _\ _\ _)) \end{cases}$$
907
908
909 *Instruction Rule.* checks the shift-of-range, division-by-zero of related immediate-related instructions, jump-out-of-
910 branch of jump instructions, and the version of instructions, etc. In particular, $check_lddw$ ensures that the $L(pc+1)$
911 bytecode has all zero for non-immediate fields to avoid invalid instructions.
912
913
914
915
916
917
918
919
920
$$verifier_ins(L, pc, sv) \stackrel{\text{def}}{=} \bigwedge \begin{cases} L[pc] = \texttt{ALU32}\ lsh/rsh/arsh\ _\ i \longrightarrow 0 \leq i \leq 31 \\ L[pc] = \texttt{ALU64}\ lsh/rsh/arsh\ _\ i \longrightarrow 0 \leq i \leq 63 \\ L[pc] = \texttt{MDM32/MDM64/PQR32/PQR64}\ op\ _\ i \longrightarrow i \neq 0 \\ L[pc] = \texttt{Ja}\ ofs \vee \texttt{JUMP}\ _\ _\ _\ ofs \longrightarrow 0 \leq pc + ofs + 1 < Length(L)/8 \\ L[pc] = \texttt{PQR32/PQR64}\ _\ _\ _ \longrightarrow sv \neq V1 \\ L[pc] = \texttt{UHMUL/SHMUL/HOR64}\ _\ _ \longrightarrow sv \neq V1 \\ L[pc] = \texttt{ADD_SP}\ i \longrightarrow sv \neq V1 \\ L[pc] = \texttt{LE}\ _\ i \longrightarrow sv = V1 \wedge i \in \{16, 32, 64\} \\ L[pc] = \texttt{NEG32/NEG64}\ _ \longrightarrow sv = V1 \\ L[pc] = \texttt{LDDW}\ _\ _\ _ \longrightarrow sv = V1 \wedge check_lddw(pc, L) \end{cases}$$
921
922
923
924
925
926
927 *Safety of Step.* Based on the formal model of the Solana verifier, we verify a simple property: the instruction rule in
928 the verifier ensures the one-step execution of the Solana interpreter is safe. That is, the **step** executes beginning from a
929 normal state, it never goes to the $Error$ state.
930
931
932 Lemma 6.4 (Step Safety). *If* $verifier_ins(L,\ pc,\ \mathcal{S}.Version) = True$, *then* $\mathbf{step}(L[pc],\ \mathcal{S}) \neq\ Err$
933
934 Proof. The proof begins by conducting a case analysis on the assembly instruction $L[pc]$, then benefits the proof
935 automation of Isabelle/HOL to solve all subgoals. □
936

6.3 Solana x86-64 JIT Compiler Proof

The Solana x86-64 JIT compiler is structured as a collection of mini-compilers, each responsible for translating an eBPF opcode into the corresponding x86-64 binary code, with an outer loop and match-pattern statement orchestrating the composition of these mini-compilers. In this paper, we focus on verifying a subset of these mini-compilers, specifically those related to ALU instructions, using our formal SBPF semantics. The full JIT implementation is highly complex, comprising over 2k lines of Rust code, including a sophisticated fuel consumption algorithm, which would require significant proof effort to verify comprehensively. We leave it for future work.

From a high-level perspective, we formalize JIT correctness using a stepwise specification approach. We model abstract machines for both SBPF and x86-64 at the binary level, where the specification defines JIT correctness as the behavioural equivalence between executing a source BPF instruction and the corresponding target instructions generated by the JIT. Each step of the SBPF abstract machine corresponds to the function '**step**' defined in subsection 4.2. For this approach, a formal binary-level semantics of the x86-64 ISA is required.

x86-64 Binary Semantics. Closely related work of x86-64 semantics in Isabelle/HOL including Sail [Armstrong et al. 2019] and X86_Semantics in AFP [Verbeek et al. 2021] have their limitations: Sail uses bit-level operations to formalize each x86-64 instruction and it complicates all subsequent proofs required for verifying the correctness of Solana JIT as SBPF's semantics is based on word-level operations. Meanwhile, the X86_Semantics in AFP only formalized a small subset of basic x86 instructions at the assembly level, insufficient for the comprehensive needs of the Solana JIT proof, and inadequate to fully bridge the gap between binary and assembly semantics.

Consequently, we have developed a new formal binary x86-64 model, along with a decoder-encoder pair in Isabelle/HOL, capable of interpreting all 190 target instructions used by the SBPF x86-64 JIT compiler. We have proven that the x86-64 decoder-encoder pair satisfies the equivalence property, ensuring a bijective relationship between the binary and assembly representations. This equivalence effectively elevates the JIT correctness proof from the x86-64 binary level to the assembly level, thus profoundly simplifying the verification process. This x86-64 abstract machine defines the program state $\mathcal{S}_{x64} ::= \langle \mathcal{R}_{x64}, \mathcal{M} \rangle$, consisting of a memory model (identical to subsection 4.2) and a x86-64 register map, which associates the 16 x86-64 integer registers with their respective values. The machine's transition is defined as $\mathcal{S}_{x64} \xrightarrow{\text{ins}} \mathcal{S}'_{x64}$, representing the execution of one x86-64 instruction and the resulting state transition from $\mathcal{S}_{x64}$ to $\mathcal{S}'_{x64}$. Similarly, we use $\mathcal{S}_{x64} \xrightarrow{l} \mathcal{S}'_{x64}$ to represent the sequential execution of a list of instructions l.

Mini-JIT Proof: ALU. We begin by introducing the function $f_{reg} : R_{SBPF} \rightarrow R_{x64}$, which, faithful to the original code, maps each SBPF register to its corresponding x86-64 register.

$$
\begin{aligned}
&f_{reg}(r) \overset{\text{def}}{=} \textbf{match } r \textbf{ with} \\
&R0 \Rightarrow rax \mid R1 \Rightarrow rsi \mid R2 \Rightarrow rdx \mid R3 \Rightarrow rcx \mid R4 \Rightarrow r8 \mid R5 \Rightarrow r9 \mid R6 \Rightarrow r12 \mid R7 \Rightarrow r13 \mid \\
&R8 \Rightarrow r14 \mid R9 \Rightarrow r15 \mid FP \Rightarrow rbp \mid PC \Rightarrow rip
\end{aligned}
$$

Leveraging the decoder-encoder equivalence proof for the x86-64 architecture, the mini JIT compiler for individual SBPF ALU instructions can be abstracted as the function $jit_{ins} : ins_{SBPF} \rightarrow ins_{x64}\ list$, which translates each SBPF ALU instruction into a list of x86-64 instructions according to JIT compilation rules. For example, the SBPF 64-bit addition between registers is mapped to the x86-64 64-bit addition instruction `addq`, with the registers appropriately mapped

using f_{reg}. Similarly, the SBPF 64-bit addition with an immediate value is translated into two x86-64 instructions: a 64-bit move instruction `movq` to load the immediate value into the temporary register $r10$, followed by an `addq` instruction.

$$
\begin{aligned}
& jit_{mini}(ins) \stackrel{\text{def}}{=} \textbf{match } ins \textbf{ with} \\
& \quad \texttt{ALU64}\ add\ r_d\ r_s \Rightarrow [\texttt{addq}\ f_{reg}(r_d)\ f_{reg}(r_s)] \mid \\
& \quad \texttt{ALU64}\ add\ r_d\ i \Rightarrow [\texttt{movq}\ r10\ i;\ \texttt{addq}\ f_{reg}(r_d)\ r10] \mid \\
& \quad \ldots
\end{aligned}
$$

The correctness of the mini JIT compiler is defined as a forward simulation between two abstract machines for proof simplification, because one SBPF instruction may be translated to many x86-64 instructions.

Definition 6.5 (Mini-JIT Correctness). A JIT compiler emits correct target instructions for a given SBPF ALU instruction if the execution of the emitted instructions results in a target state that preserves the simulation relation $\sim$.
$\forall ins.\ \mathcal{S}_{SBPF} \xrightarrow{ins} \mathcal{S}'_{SBPF} \wedge \mathcal{S}_{SBPF} \sim \mathcal{S}_{x64} \Longrightarrow \exists\, \mathcal{S}'_{x64}.\ \mathcal{S}_{x64} \xrightarrow{jit_{mini}(ins)} \mathcal{S}'_{x64} \wedge \mathcal{S}'_{SBPF} \sim \mathcal{S}'_{x64}$

Given that the ALU instructions only affect register values, we can establish the relation $\sim$ as a direct correspondence between the register value in the SBPF state and the mapped register in the x86-64 state, *i.e.,* $\sim \stackrel{\text{def}}{=} \forall\, r.\ [[r]] = [[f_{reg}(r)]]$.

7 Evaluation

This section evaluates our formalization effort and clarifies the limitations of the methodology proposed in this paper.

7.1 Implementation

As there is no official counting tool for Isabelle/HOL, we use a common 'Count Lines of Code' tool named CLoC to count Isabelle/HOL, Rust, and OCaml implementation. In particular, we use the flag '–force-lang="OCaml"' to count our Isabelle/HOL code because both Isabelle/HOL and OCaml use '(* *)' for comments.

Formal Verification. Table 2 shows lines-of-code statistics. The main modules (*i.e.,* assembler, disassembler, verifier, and interpreter) of Solana VM consist of around 2k lines of code in Rust and the Solana x86-64 JIT compiler has more than 2k lines of Rust implementation. The Isabelle/HOL development comprises about 2.7k specifications of the main components of Solana VM, plus around 0.4k lines of proof. As the Solana JIT is quite complex, we only formalize a small part of the Solana JIT, around 1k lines of specification along with more than 1.8k lines of proof.

The entire verification and validation effort took around 11 person-months, of which 60% were spent on the SBPF verification, 30% on x86-64 binary formalization, 10% on the validation framework. We spent a lot of verification effort on the target x86-64 language of the Solana JIT compiler, including about 2.6k lines of specification and 5k lines of proof, because the Solana x86-64 JIT compiler generates the binary code using various x86-64 encoding patterns, plus the complexity of CISC-styled instruction formats.

Validation Framework. Our validation framework is implemented using a combination of OCaml and Rust. It includes approximately 100 lines of OCaml code for data format processing and around 600 lines for random instruction generation and testing. To address the limitations of the existing Isabelle/HOL extraction mechanism, our lightweight solution requires only minimal adjustments: the Isabelle/HOL glue code consists of three functions totalling 11 lines, while the OCaml glue code is composed of five functions, comprising 20 lines in total.

Component	Language	Lines (KLOC)	Effort (.pm)
Solana VM (excluded JIT) + JIT Compiler	Rust	2.0 + 2.1	
SBPF Syntax	Isabelle/HOL	0.9	0.5
SBPF Semantics	Isabelle/HOL	0.9	1.5
SBPF Verifier + Safety	Isabelle/HOL	0.2 + 0.1	1
SBPF Assembler-Disassembler + Consistency	Isabelle/HOL	0.7 + 0.3	1
SBPF JIT	Isabelle/HOL	1.0	1
SBPF JIT proof	Isabelle/HOL	1.8	1
SBPF x86-64 Specification	Isabelle/HOL	2.6	1.5
SBPF x86-64 Proof	Isabelle/HOL	5.0	2.5
SBPF Validation Framework	OCaml + Rust	0.1 + 0.6	1
SBPF Executable Semantics	OCaml	4.9	

Table 2. Code and proof statistics. pm stands for person-months.

For comparison, the extracted code of SBPF executable semantics is about 5k lines of OCaml implementation, significantly more complex than both the Isabelle/HOL specification and the original Rust implementation. This is primarily due to two factors: first, the extracted code includes numerous definitions of basic types and associated operations, *e.g.,* natural numbers *nat* and integers *int*. Second, the extracted code follows a constructive style, which, while correct, is less human-readable. For example, the Isabelle/HOL implementation of the **interpreter** function spans 19 lines, while the corresponding generated OCaml code expands to 53 lines.

Report to the Solana Community. During the formalization of SBPF semantics, we discovered a potential issue that could lead to illegal behaviour in the Solana interpreter and JIT compiler. Specifically, this issue allowed Solana_v2-specific instructions, such as `LDDW` and `HOR64`, to be executed across all VM versions, bypassing version constraints. The root cause was the absence of version-checking mechanisms in both the interpreter and JIT.

```
//verifier.rs
ebpf::HOR64_IMM  if !sbpf_version.enable_lddw() => {},

//interpreter.rs or jit.rs
ebpf::HOR64_IMM  => { /* the wrong version: lost version checking  */
/* the correct version:
ebpf::HOR64_IMM if !self.executable.get_sbpf_version().enable_lddw() => {  */
```

Additionally, our verification of the Solana assembler-disassembler pair identified another issue: the source register index range constraint was missing, which compromised the consistency property.

Both issues, along with proposed fixes, were reported to the Solana community, where they were confirmed and have since been addressed in the latest release.

7.2 Lessons Learned

We clarify the prospects and limitations of the methodology proposed in the paper and its application to the Solana VM.

Our Goal. The methodology aims at providing the first and most complete semantics foundation for the SBPF ISA in Isabelle/HOL. It is the main contribution of this paper.

External Calls. While we formalize the internal function call mechanism using the function call map, our SBPF semantics does not include a formalization of Solana's external system APIs. We choose to trust these APIs, as they are

less prone to errors. The external APIs include three printing functions (which print the last three arguments, print a string, and print five arguments in hexadecimal format, respectively), one function for aggregating five arguments into a single 'u64', and two functions, `memfrob` and `strcmp`, both originating from the C programming language.

Application Limitations. Regarding its application to the Solana eBPF VM, we first provide a semantics validation framework to relax the gap between our high-level specification and the original low-level Rust implementation. Our benchmarks reuse the test suite of Solana but exclude the system call cases.

Then we apply our semantics to formalize the main modules of the Solana VM and prove some key properties of those modules. For the Solana x86-64 JIT compiler, we only provide the formalism of binary semantics of the target x86-64 ISA and prove that some ALU instructions could be compiled correctly into the target binary code. We made this choice as the code size of the JIT compiler is comparable to the rest components of Solana but the level of complexity is much higher than the rest, and be another verification project in its own right.

Extraction Limitations. Finally, we trust the Isabelle/HOL extraction mechanism. The Isabelle/HOL extractor could go wrong to render the final object code incorrect, but its correctness is beyond our scope. The alternative solution is to present a compiler to either extract Rust from Isabelle/HOL specification or translate Rust to Isabelle/HOL. Proving the correctness of such a compiler in Isabelle/HOL would also be a non-trivial verification task.

However, we believe that these two last limitations could be relaxed once we complete the verification of SBPF x86-64 JIT compiler and the Rust2Isabelle/HOL transilper. Essentially, the last mile of our journey toward two complete compilers would reuse our x86-64 binary semantics to formalize the rest part of x86-64 JIT compiler and develop a trusted deep embedding way to express Rust code in Isabelle/HOL.

8 Related Work

Many related projects have hosted a formal semantics of eBPF as their main contribution or as part of their infrastructure. This section reviews research efforts related to our approach and compares it to our formal semantics based on three directions that reflect the primary contributions of our work: the completeness of ISA, the entire VM application, and the validation gap.

The Jitk framework [Wang et al. 2014] uses Coq to implement and verify the correctness of a JIT compiler for the classic Berkeley Packet Filter language (not eBPF) in the Linux kernel. Jitk translates the BPF bytecode into CompCert and leverages the CompCert backend to generate target code. The classic BPF ISA of Jitk is much more limited than ours. It only has two registers and describes the semantics of 43 instructions, while our semantics covers all 116 instructions in the SBPF ISA. The Jitk compiler is extracted to OCaml implementation using the Coq extraction mechanism. Our executable code uses the Isabelle/HOL extraction, similar to Coq, and we additionally provide sufficient validation to enhance the confidence of our formal semantics.

JitSynth [Van Geffen et al. 2020] is a tool designed for synthesizing verified JITs for in-kernel DSLs, and it has been applied to synthesize a JIT compiler from eBPF to RISC-V. JitSynth only considers a subset of eBPF ISA (87 of the 115 instructions), their work doesn't aim for completeness of semantics, one of our primary goals.

Serval [Nelson et al. 2019] is a framework that enables scalable verification for systems code via symbolic evaluation. It formalizes the semantics of the eBPF ALU instructions in Rosette [Torlak and Bodik 2013], a solver-aided programming language for program synthesis and verification based on symbolic execution. The Serval framework includes a checker for eBPF JIT compilers to verify the determinism property of JIT compilers.

Jitterbug [Nelson et al. 2020] provides a framework with a specification of JIT correctness and generates automated proofs for various real-world Linux eBPF JIT compilers. Jitterbug extends Serval to support the semantics of a large subset of eBPF ISA in Rosette. Jitterbug mainly focuses on the JITs components of eBPF, and there is also no verification or validation between its formal JIT model and the extracted unverified C code.

K2 [Xu et al. 2021] is a compiler that optimizes eBPF bytecode with formal correctness and safety guarantees. It currently only handles a subset of eBPF ISA, including ALU instructions, memory instructions, and eBPF call instructions.

PREVAIL [Gershuni et al. 2019] is an eBPF verifier based on abstract interpretation implemented in C++, which supports more program structures, such as loops, and efficiently outperforms the standard Linux verifier. It supports most of the eBPF ISA features (except for the internal calls, etc.), and there is no mechanized semantics for the eBPF ISA.

A formal range analysis of the Linux eBPF verifier is proposed [Sanjit and Hovav 2023], but it only considers most arithmetic instructions and doesn't discuss the arithmetic multiplication. Agni [Vishwanathan et al. 2023] is another formal range analysis of the eBPF verifier, and it provides the semantics of all ALU and jump instructions of the eBPF ISA. For the soundness proof of the range analysis, Agni generates the first-order logic formula from the verifier's C source code and uses the Z3 SMT solver [de Moura and Bjørner 2008] for checking formulas.

To the best of our knowledge, the most closely related work is the CertrBPF project [Yuan et al. 2022, 2023; Zandberg et al. 2022], which introduces a formally verified eBPF VM for the IoT operating system RIOT-OS, developed in Coq, with an equivalent C implementation extracted using an end-to-end verification workflow. CertrBPF formalizes all instructions of RIOT-OS eBPF, a variant of eBPF that includes a substantial subset of the Linux eBPF ISA, and proves the safety of both the verifier and the interpreter. A follow-up work, CertrBPF-JIT [Yuan et al. 2024], extends this by providing a verified JIT compiler for the RIOT-OS eBPF VM, also formalized in Coq. However, this work is currently limited to the ARM architecture and supports only a small subset of arithmetic instructions. In contrast to CertrBPF, our work offers the first complete formal semantics of the SBPF ISA and further formalizes the assembler-disassembler pair, accompanied by a consistency proof. Both CertrBPF and our work trust the external calls. While CertrBPF presents an innovative end-to-end theorem from Coq to C, it doesn't discuss if its Coq formalization faithfully describes the behaviours of the original implementation. Our approach includes extensive testing to validate our Isabelle/HOL model, addressing this gap in CertrBPF's methodology.

9 Conclusion

In this paper, we have presented the first complete formal semantics of Solana eBPF binary instructions to date, and have thoroughly validated it using a novel testing framework and applied it by formalizing several Solana components along with the proofs of the key properties. All have been mechanically verified in Isabelle/HOL.

We are carrying on the following research:

Formally verified JIT compiler of SBPF. The Solana eBPF VM includes a x86-64 JIT compiler that translates all SBPF instructions into x86-64 binary. We are formalizing the whole JIT compiler in Isabelle/HOL, and the next step is to prove the semantics preservation theorem of this JIT compiler. One of the most challenging parts is to prove the correctness of the compute units consumption algorithm.

Rust2Isabelle/HOL verified compiler. The Solana eBPF VM is implemented in Rust, to fill the gap between Rust and Isabelle/HOL, one way is to design a code generator to deeply embed the Rust (intermediate) representation into Isabelle/HOL definition in a syntax-directed manner, then to give a verified lifting from this deep embedding syntax to high-level Isabelle/HOL specification.

1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248

Data-Availability Statement

The source code and proofs of our work, its generated code and benchmark data are available to the OOPSLA'25 review committee on an anonymized repository [Anonymised 2024]. We clarify:

- *Nature*: The artifact includes
  - *Isabelle/HOL code*: the semantics model of SBPF ISA, the formalization of SBPF verifier, assembler, disassembler, interpreter, (part) JIT compiler, x86-64 binary semantics and related proofs.
  - *OCaml + Rust code*: the extracted executable semantics in OCaml, and the original Solana eBPF VM in Rust, and our validation framework in Rust.
  - *Makefile code*: a makefile script to start the Isabelle/HOL project, validate semantics, and perform code and proof statistics.
- *Limitations*: The artifact is a zip file instead of an entire VM.
  - *Libraries dependent*: It relies on some basic libraries which require users to install them manually, *e.g.,* the Isabelle/HOL software, the OCaml environment, and the Count lines of code tool 'CLoC', etc.
  - *Support OS*: We only test our artifact on two OS environments.
    - Windows 11 + WSL2 (Ubuntu 22.04 LTS)
    - Ubuntu 22.04 LTS
- *Artifact Evaluation*: Our artifact will be submitted for Artifact Evaluation.

References

Elvira Albert, Samir Genaim, Daniel Kirchner, and Enrique Martin-Martin. 2023. Formally Verified EVM Block-Optimizations. In *Computer Aided Verification*, Constantin Enea and Akash Lal (Eds.). Springer Nature Switzerland, Cham, 176–189.

Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 66–77. https://doi.org/10.1145/3167084

Anonymised. 2024. A complete formal semantics of eBPF instruction set architecture for Solana VM. https://anonymous.4open.science/r/SBPF-EC62/

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. *Proc. ACM Program. Lang.* 3, POPL, Article 71 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290384

BoredPerson. 2024. Fix JIT second level defence. https://github.com/solana-labs/rbpf/pull/557

Franck Cassez, Joanne Fuller, Milad K. Ghale, David J. Pearce, and Horacio M. A. Quiles. 2023. Formal and Executable Semantics of the Ethereum Virtual Machine in Dafny. In *Formal Methods*, Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker (Eds.). Springer International Publishing, Cham, 571–583.

Siwei Cui, Gang Zhao, Yifei Gao, Tien Tavu, and Jeff Huang. 2022. VRust: Automated Vulnerability Detection for Solana Smart Contracts. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 639–652. https://doi.org/10.1145/3548606.3560552

Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. 2019. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1133–1148. https://doi.org/10.1145/3314221.3314601

Jeremy Dawson. 2009. Isabelle Theories for Machine Words. *Electronic Notes in Theoretical Computer Science* 250, 1 (2009), 55–70. https://doi.org/10.1016/j.entcs.2009.08.005 Proceedings of the Seventh International Workshop on Automated Verification of Critical Systems (AVoCS 2007).

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

Dxo, Mate Soos, Zoe Paraskevopoulou, Martin Lundfall, and Mikael Brockman. 2024. Hevm, a Fast Symbolic Execution Framework for EVM Bytecode. In *Computer Aided Verification*, Arie Gurfinkel and Vijay Ganesh (Eds.). Springer Nature Switzerland, Cham, 453–465.

Matt Fleming. 2017. A Thorough Introduction to eBPF.

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* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1069–1084. https://doi.org/10.1145/

3314221.3314590

Sudhanshu Goswami. 2005. An introduction to KProbes. https://lwn.net/Articles/132196/

E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, Los Alamitos, CA, USA, 204–217. https://doi.org/10.1109/CSF.2018.00022

Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *Financial Cryptography and Data Security*, Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson (Eds.). Springer International Publishing, Cham, 520–535.

Meta Incubator. 2018. A high performance layer 4 load balancer. https://github.com/facebookincubator/katran

Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. 2020. Semantic Understanding of Smart Contracts: Executable Operational Semantics of Solidity. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1695–1712. https://doi.org/10.1109/SP40000.2020.00066

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. http://xavierleroy.org/publi/compcert-CACM.pdf

Ximeng Li, Zhiping Shi, Qianying Zhang, Guohui Wang, Yong Guan, and Ning Han. 2019. Towards Verifying Ethereum Smart Contracts at Intermediate Language Level. In *Formal Methods and Software Engineering*, Yamine Ait-Ameur and Shengchao Qin (Eds.). Springer International Publishing, Cham, 121–137.

Andreas Lochbihler. 2018. Fast Machine Words in Isabelle/HOL. In *Interactive Theorem Proving*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer International Publishing, Cham, 388–410.

Diego Marmsoler and Achim D. Brucker. 2021. A Denotational Semantics of Solidity in Isabelle/HOL. In *Software Engineering and Formal Methods*, Radu Calinescu and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 403–422.

Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Usenix Winter Conference*, Vol. 46. USENIX, San Diego, California, USA, 259–270.

Microsoft. 2019. eBPF implementation that runs on top of Windows. https://github.com/microsoft/ebpf-for-windows

Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 225–242. https://doi.org/10.1145/3341301.3359641

Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, USA, 41–61. https://www.usenix.org/conference/osdi20/presentation/nelson

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg.

Daejun Park, Yi Zhang, and Grigore Rosu. 2020. End-to-End Formal Verification of Ethereum 2.0 Deposit Smart Contract. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 151–164.

Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. 2018. A formal verification tool for Ethereum VM bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 912–915. https://doi.org/10.1145/3236024.3264591

Bhat Sanjit and Shacham Hovav. 2023. Formal Verification of the Linux Kernel eBPF Verifier Range Analysis. https://sanjit-bhat.github.io/assets/pdf/ebpf-verifier-range-analysis22.pdf

Kudelski Security. 2019. Solana Labs Architectural Security Review and Report. https://kudelskisecurity.com/wp-content/uploads/Solana-LabsArchitectural-Security-Review-andReport.pdf

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. https://doi.org/10.1145/1785414.1785443

Sven Smolka, Jens-Rene Giesen, Pascal Winkler, Oussama Draissi, Lucas Davi, Ghassan Karame, and Klaus Pohl. 2023. Fuzz on the Beach: Fuzzing Solana Smart Contracts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1197–1211. https://doi.org/10.1145/3576915.3623178

Solana-labs. 2018. solana rbpf. https://github.com/solana-labs/rbpf

Solana-labs. 2024. Fix callx. https://github.com/solana-labs/rbpf/pull/583

Dave Thaler. 2024. BPF Instruction Set Architecture (ISA) draft-ietf-bpf-isa-04. https://datatracker.ietf.org/doc/draft-ietf-bpf-isa/

Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

Jacob Van Geffen, Luke Nelson, Isil Dillig, Xi Wang, and Emina Torlak. 2020. Synthesizing JIT Compilers for In-Kernel DSLs. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 564–586.

Freek Verbeek, Abhijith Bharadwaj, Joshua Bockenek, Ian Roessle, Timmy Weerwag, and Binoy Ravindran. 2021. X86 instruction semantics and basic block symbolic execution. https://isa-afp.org/entries/X86_Semantics.html, Formal proof development.

Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2023. Verifying the Verifier: eBPF Range Analysis Verification. In *Computer Aided Verification*, Constantin Enea and Akash Lal (Eds.). Springer Nature Switzerland, Cham, 226–251.

Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. 2014. Jitk: A Trustworthy In-Kernel Interpreter Infrastructure. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 33–47. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/wang_xi

Yuepeng Wang, Shuvendu Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. 2019. Formal Specification and Verification of Smart Contracts for Azure Blockchain. https://www.microsoft.com/en-us/research/publication/formal-specification-and-verification-of-smart-contracts-for-azure-blockchain/

Qiongwen Xu, Michael D. Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. 2021. Synthesizing safe and efficient kernel extensions for packet processing. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 50–64. https://doi.org/10.1145/3452296.3472929

Shenghao Yuan, Frédéric Besson, and Jean-Pierre Talpin. 2024. End-to-End Mechanized Proof of a JIT-Accelerated eBPF Virtual Machine for IoT. In *Computer Aided Verification*, Arie Gurfinkel and Vijay Ganesh (Eds.). Springer Nature Switzerland, Cham, 325–347.

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 *Computer Aided Verification*, Sharon Shoham and Yakir Vizel (Eds.). Springer International Publishing, Cham, 293–316.

Shenghao Yuan, Benjamin Lion, Frédéric Besson, and Jean-Pierre Talpin. 2023. Making an eBPF Virtual Machine Faster on Microcontrollers: Verified Optimization and Proof Simplification. In *Dependable Software Engineering. Theories, Tools, and Applications*, Holger Hermanns, Jun Sun, and Lei Bu (Eds.). Springer Nature Singapore, Singapore, 385–401.

Koen Zandberg, Emmanuel Baccelli, Shenghao Yuan, Frédéric Besson, and Jean-Pierre Talpin. 2022. Femto-Containers: Lightweight Virtualization and Fault Isolation for Small Software Functions on Low-Power IoT Microcontrollers. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference* (Quebec, QC, Canada) *(Middleware '22)*. Association for Computing Machinery, New York, NY, USA, 161–173. https://doi.org/10.1145/3528535.3565242

Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark Barrett, and David L. Dill. 2020. The Move Prover. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 137–150.