





Formalizing x86-64 ISA in Isabelle/HOL: A Binary Semantics for eBPF JIT Correctness

Jiayi Lu¹, Shenghao Yuan² (✉), David Sanan³, and Yongwang Zhao^{1,2}

¹ School of Cyber Science and Technology, and College of Computer Science and Technology, Zhejiang University, Hangzhou, China

² State Key Laboratory of Blockchain and Data Security, Zhejiang University, China

³ InfoComm Technology Cluster, Singapore Institute of Technology, Singapore
shenghaoyuan0928@163.com

Abstract. Binary semantics forms a critical infrastructure for proving the correctness of Just-In-Time (JIT) compilation, an advanced technique to optimize runtime execution by translating the source bytecode of a program into target machine code. This paper presents a formal model of x86-64 binary semantics in Isabelle/HOL, as the foundation to verify the eBPF x86-64 JIT correctness for the Solana blockchain. First, we formalize a significant subset of x86-64 semantics, covering all the x86-64 instructions used in the Solana JIT. Second, we develop an encoder and decoder to establish a precise bidirectional mapping between assembly and binary code. Furthermore, we demonstrate the equivalence property of the x86-64 encoder-decoder pair, significantly simplifying the correctness proof of the Solana JIT compiler verification.

Keywords: x86-64 · ISA · Binary Semantics · eBPF Virtual Machine · Solana blockchain · Isabelle/HOL · Formal Verification.

1 Introduction

As Paul Tyma stated in 1998, before the use of Just-In-Time (JIT) compilation, ‘Java is slow. Java isn’t just slow, it’s really slow, surprisingly slow’ [22]. JIT compilation has since emerged as an effective technique for speeding up program interpreters by dynamically translating source bytecode into target binary code, which is then executed as native machine code. This technique is now widely used in various interpreter-based implementations of programming languages, from Java to Linux eBPF [5] and its variants, specifically, the Solana blockchain eBPF virtual machine [21]. Given the widespread deployment of eBPF JITs in safety-critical environments, ensuring their correctness is essential for system security. Vulnerabilities within these JIT compilers could significantly compromise the security of Linux kernels or affect millions of transactions on the Solana platform.

Formal verification, particularly through theorem provers like Coq [3] and Isabelle/HOL [17], offers a rigorous approach to mathematically ensuring the correctness of eBPF JITs. However, compared to standard compiler verification (*e.g.*, CompCert [12] that verifies the compilation from a subset of C to various

target assembly languages), to verify a target-specific eBPF JIT is considerably more challenging due to the intricate nature of low-level binary semantics.

In the context of the Solana JIT, the main theorem for proving its correctness involves maintaining simulation preservation (see [subsection 2.1](#) for details) between the source eBPF and the target x86-64 binary-level semantics. Essential to this proof is a formalized x86-64 binary semantics that incorporates the encoder-decoder equivalence property. This ensures the exact bijection between the binary and assembly representation, and the simulation to be conducted at the assembly level, thus profoundly simplifying the verification process.

However, existing research has yet to fully address this demand. Closely related work in Isabelle/HOL includes Sail [1] and X86_Semantics in AFP [23]. The Sail model translates an ACL2 [18] formal model of x86-64 ISA into its domain-specific language and produces corresponding Isabelle/HOL scripts. Nevertheless, the extensive use of bit-level semantics that Sail utilizes to formalize each x86-64 instruction noticeably complicates all subsequent proofs required for verifying the correctness of Solana JIT. Additionally, Sail lacks x86-64 encoding rules for proving encoder-decoder equivalence which is essential for streamlining the Solana JIT proofs. 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.

Challenges To establish a formal binary x86-64 model for the Solana eBPF JIT presents two major challenges:

- *Binary and Assembly Semantics Gap.* In contrast to the assembly level, binary semantics contains more complicated machine-level information. For example, a single assembly mnemonic can map to multiple, even exceeding ten, binary encodings (see [subsection 4.3](#)). Therefore, an effective formalization of binary semantics requires a bijective encoder-decoder pair for accurate machine execution. All the vital information (*e.g.*, operands, memory addresses, and instruction lengths) must be preserved throughout the bijection process. Furthermore, leveraging existing models like Sail poses challenges as it potentially increases the overall efforts to prove JIT correctness.
- *Encoding Complexity of the x86-64 ISA.* The x86-64 architecture is notoriously complicated [4,15,27]. The precise modeling of this architecture is critical, but the intricate CISC design demands a meticulous approach to handle properties such as variable instruction length, diverse bit-mode representations, and backward compatibility with legacy instructions. For the Solana JIT which targets a broad subset of the x86 ISA, this encoding complexity makes formalizing the decoder particularly demanding.

Contributions Since existing models in Isabelle/HOL do not capture the semantics needed to necessarily support Solana eBPF x86-64 JIT verification, we have formalized a new binary x86-64 model capable of interpreting the x86-64 binary code generated by the Solana JIT. To establish confidence in our model,

we base the formalization on the CompCert x86-64 semantic framework and enhance it by extending the capabilities to support more instructions. We choose Isabelle/HOL over other systems (*e.g.*, Coq or SMT solvers) to implement our model, due to its minimal trusted computing base (TCB) and the high degree of proof automation. In essence, this paper introduces the first binary formal semantics that encompasses Solana JIT implementation, along with an equivalence property demonstrated to simplify the JIT correctness proof. Our formalization lays the groundwork for a complete certification of the Solana eBPF JIT compiler. The contributions of this work are summarized as follows:

- *Formalization of x86-64 Binary Semantics.* We have formalized the semantics for all the necessary x86-64 instructions utilized by the Solana JIT (*i.e.*, 78 semantic definitions covering 190 assembly instructions, which correspond to 303 binary representations). Then we developed a formal model of the encoder-decoder pair, establishing an accurate bijection between the assembly and binary code. Our models are faithful to the behaviors of the instructions in the Intel 64 documentation, ensuring direct support for real-world scenarios.
- *Verification of Encoder-Decoder Equivalence.* We have provided the equivalence property of the x86-64 decoder-encoder pair, which distinguishes our work from existing research. This property guarantees that the binary semantics can be lifted up to the assembly level, thereby significantly streamlining the proof effort required for the main theorem of the Solana JIT correctness.

All the formalization and proofs have been mechanized in Isabelle/HOL (see [14]), comprising approximately 8k lines of code total. This includes around 2.8k lines of specification for the x86-64 binary semantic model and 5.2k lines of proof for the encoder-decoder equivalence property.

The paper is organized as follows: [Section 2](#) provides background materials. [Section 3](#) proposes the semantic formalization of x86-64 ISA subset. [Section 4](#) presents the encoder and decoder model and their equivalence proof is given in [Section 5](#). [Section 6](#) introduces related works and [Section 7](#) concludes.

2 Preliminaries

This section first presents an overview of the Solana JIT correctness, addressing the foundational importance of binary semantics in this proof. Second, it introduces the x86-64 machine instruction format.

2.1 Solana JIT Correctness and x86-64 Binary Semantics Overview

eBPF (Extended Berkeley Packet Filter) originated for network packet filtering and has evolved to run sandboxed programs within the Linux kernel, enhancing its capabilities without modifying the kernel. Solana, an innovative blockchain platform, leverages Linux eBPF to implement its VM for executing Solana smart

contracts. One key component of Solana eBPF VM is its JIT compiler that translates eBPF bytecode to x86-64 binaries to optimise performance.

The Solana eBPF JIT workflow is shown in [Figure 1](#). Consider the example of a user-provided eBPF binary instruction, $ins = 0x0CXY000000000000$, where X and Y are hexadecimal digits: 1) The instruction is decoded into the eBPF assembly $BPF_ADD32\ R_X\ R_Y$ where $0x0C$ is the eBPF opcode of a 32-bit addition, and X/Y is the indices of the destination/source register; 2) Under the Solana per-instruction JIT rule (JIT_{rule}), two corresponding x86-64 assembly instructions are generated: one for a 32-bit addition and another for extending the result from 32-bit to 64-bit. The x86-64 $r_{X'}$ and $r_{Y'}$ are determined by a specific mapping function $fr : R_{BPF} \rightarrow r_{x64}$ that translates each eBPF register into a corresponding x86-64 register; 3) x86-64 instructions are encoded to L , a set of x86-64 binary instructions of dynamic sizes, using a Solana-specific encoding algorithm. This sequence, L , is referred to as the jited x86-64 code.



Fig. 1. Solana eBPF x86-64 JIT compiler workflow (for BPF_ADD_{32} instruction)

To prove the correctness of this JIT workflow, the main theorem posits a forward simulation, as shown in [Figure 2](#) (left). This theorem guarantees that each execution step ($step_{BPF}$) of the input eBPF binary code can be simulated by one or more execution steps ($step_{x64}^+$) of the output jited x86-64 code. The correctness of this simulation is established through value equality between each eBPF register R_i and its corresponding x86-64 register as determined by the mapping function fr , formally defined as $\approx \stackrel{\text{def}}{=} \forall i, \llbracket fr(R_i) \rrbracket = \llbracket R_i \rrbracket$. Given that the JIT performs binary translation and takes $step$ at the assembly level, two distinct decoder functions are utilized to handle the respective instruction sets.

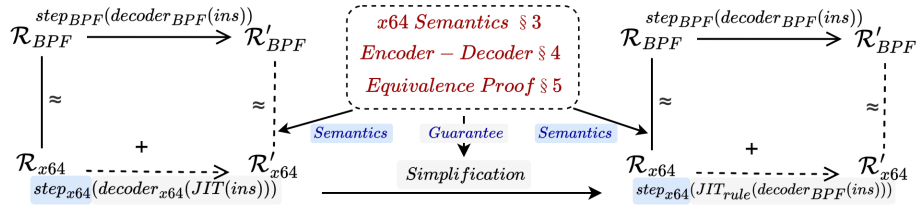


Fig. 2. Solana eBPF JIT simulation proof with the x86-64 binary target (left), our x86-64 binary semantics as an infrastructure for the Solana eBPF simulation proof (mid), and the simplified simulation proof with the x86-64 assembly target (right).

[Figure 2](#) also illustrates how this paper builds the infrastructure that forms the foundation for proving the correctness of the Solana JIT. [Section 3](#) and

Section 4 work together to provide binary semantics for the forward simulation to execute $step_{x64}$ and **Section 5** guarantees a critical simplification process in this theorem by the encoder-decoder equivalence proof. This proof simplification is conducted as follows: The (per-*instruction*) JIT compiler is defined as a composition of functions: $JIT(ins) \stackrel{\text{def}}{=} encoder_{x64}(JIT_{rule}(decoder_{BPF}(ins)))$. The x86-64 binary model guarantees the equivalence property of the encoder and decoder, which simplifies $decoder_{x64}(JIT(ins))$ (**Figure 2** bottom left) into $JIT_{rule}(decoder_{BPF}(ins))$ (**Figure 2** bottom right). This simplification effectively elevates the JIT correctness proof from the x86-64 binary level to the assembly level.

2.2 x86-64 Machine Instruction Format

x86-64 ISA includes 16 general-purpose 64-bit registers, several special-purpose registers (*e.g.*, program counter), and a variety of instructions for arithmetic, control flow, system operations, etc. All formalized instructions conform to subsets of the general machine instruction format depicted in **Figure 3**. Considering the complexity of the full x86-64 ISA and the redundancy of many components, we specifically focus on aspects of the x86-64 ISA employed by the Solana JIT.

Legacy Prefix (optional, 1-2 bytes)	REX Prefix (optional, 1 byte)	Opcode (mandatory, 1-3 bytes)	ModR/M (optional, 1 byte)	SIB (optional, 1 byte)	Displacement (optional, 1/2/4 bytes)	Immediate (optional, 1/2/4/8 bytes)
---	-------------------------------------	-------------------------------------	---------------------------------	------------------------------	--	---

Fig. 3. x86-64 general machine instruction format

- *Legacy Prefix*: An optional 1-byte code divided into four groups, each with a set of allowable prefix codes. The Solana JIT only uses the operand-size override prefix (0x66) to specify a 16-bit operand size, ignoring others.
- *REX Prefix*: An optional 1-byte code ‘0100 WRXB’ that enables 64-bit operands and accesses extended registers R8 to R15. W allows 64-bit operand size, R extends the ModR/M **reg** field, X extends the SIB **index** field, and B extends the ModR/M **r/m** field, SIB **base** field, or opcode **reg** field.
- *Opcode*: A mandatory code that defines instruction operations, consisting of an optional 1/2-byte escape prefix and a primary 1-byte opcode. The last bit of the primary opcode(w) indicates an 8-bit operand size when unset. The primary opcode can be extended by the ModR/M **reg** field.
- *ModR/M*: A 1-byte code following the opcode, specifying a register and/or memory operand. It includes a 2-bit **mod** field (for specifying the addressing mode), a 3-bit **reg** field (for the first operand or opcode extension), and a 3-bit **r/m** field (for the second operand).
- *SIB*: A 1-byte code required by certain ModR/M byte encodings to enhance memory addressing. It specifies a 2-bit **scale** factor, a 3-bit **index** register, and a 3-bit **base** register.
- *Address Displacement*: Included when the addressing mode requires a displacement, available in sizes of 1, 2, or 4 bytes.

- *Immediate data*: Included when the instruction calls for an immediate value, which can be 1, 2, 4, or 8 bytes, and is always the last field of the instruction.

Example Figure 4 illustrates the encoding of the 16-bit **addw_{r,i} R8 imm** instruction: adding the *immediate* value (assuming 0x1234) to the register R8 (0b1000).

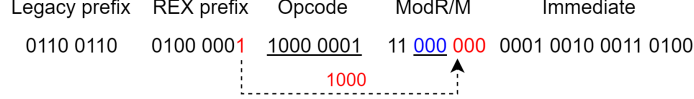


Fig. 4. Binary representation of **addw_{r,i} R8 imm**

This encoding begins with the legacy prefix (0x66). Since *R8* is an extended register, REX.B is activated and the *r/m* is set to 0b000 (colored in red). The opcode for this operation is 0x81, extended by the *reg* field 0b000 (colored in blue) to denote the addition of an immediate value to a register. The *mod* field is set to 0b11, indicating the *r/m* field represents a register operand, rather than a memory address. The binary instruction ends with a 16-bit immediate value.

In practice, the Solana eBPF JIT is more complex, as we have encountered a diverse array of instructions within different encoding patterns (see [subsection 4.1](#)). This diversity greatly complicates the decoding process as well.

3 Formalizing x86-64 ISA subset

To formalize the binary semantics of x86-64, our methodology first lifts x86-64 instructions from the binary level to the assembly level by the faithful *decoder_{x64}* function (see [subsection 4.3](#)). Then, it interprets the corresponding assembly code under the defined x86-64 semantics. This section formalizes the detailed syntax, machine state and semantics of the x86-64 ISA subset required by the Solana JIT. Our formalization mainly focuses on the functional aspects of the ISA by defining mathematical formulas that depict the behavior of instructions on registers and memory. Given that the Solana JIT only generates instructions with sequential behaviors, the concurrency features of the x86-64 memory model are not covered in this paper.

3.1 Syntax

Figure 5 shows the syntax of our x86-64 formal model. The *processor registers* include the program counter *pc*, 16 general-purpose 64-bit registers *r*, the status flags *f* (e.g., zero flag *ZF*, carry flag *CF*, etc), and the time-stamp counter register *tsc*. The float registers are omitted as they are not needed by the Solana JIT. Memory access is facilitated through both the address *addr* and the size of the accessed memory block *mb*. *addr* defines three addressing modes in the

Solana model: 1) a single signed immediate value d as the address displacement; 2) extends the first mode by adding a **base** register value; 3) extends the second mode by adding an **index** register value with a 2-bit **scale** factor. The test conditions η encompass comparisons such as equal(=), below($<_u$ for unsigned comparison), and less($<_s$ for signed comparison), among others.

(IntReg)	r	::=	$RAX \mid RBX \mid RCX \mid RDX \mid RSI \mid RDI \mid RBP \mid RSP$ $\mid R8 \mid R9 \mid R10 \mid R11 \mid R12 \mid R13 \mid R14 \mid R15$
(Rflags)	f	::=	$ZF \mid CF \mid PF \mid SF \mid OF$
(ProcReg)	$preg$::=	$pc \mid r \mid f \mid tsc$
(Address)	$addr$::=	$d \mid (r_{base}, d) \mid (r_{base}, r_{index}, scale, d)$
(MemBlock)	mb	::=	$M8 \mid M16 \mid M32 \mid M64$
(TestCond)	η	::=	$= \mid \neq \mid <_u \mid \leq_u \mid \geq_u \mid >_u \mid <_s \mid \leq_s \mid \geq_s \mid >_s \mid \dots$
(x64Ins)	ins	::=	$\mathbf{addl}_{rr} \ r_d \ r_s \mid \mathbf{andq}_{rr} \ r_d \ r_s \mid \mathbf{rolw}_{ri} \ r_d \ ofs \mid \dots \mid$ $\mathbf{mov}_{rm} \ r \ addr \ mb \mid \mathbf{mov}_{mr} \ addr \ r \ mb \mid \dots \mid$ $\mathbf{jcc} \ \eta \ d \mid \mathbf{call}_i \ d \mid \mathbf{call}_r \ r \mid \dots \mid$ $\mathbf{rdtsc} \mid \mathbf{nop} \mid \dots$

Fig. 5. The syntax of our x86-64 formal model

Our x86-64 model consists of four base sets of instructions.

- *Arithmetic and Logic Instructions:* We have formalized an extensive set of common arithmetic, logical, and bit manipulation instructions across various bit-width registers. For instance, \mathbf{addl}_{rr} declares a 32-bit addition of the source register r_s to the destination register r , \mathbf{andq}_{rr} defines a 64-bit logical AND between r_d and r_s , and \mathbf{rolw}_{ri} formalizes a 16-bit left rotate on r_d using an 8-bit shift offset ofs ;
- *Data Transfer Instructions:* We have defined necessary data transfer and conversion instructions executed by \mathbf{mov} and its variants. For instance, \mathbf{mov}_{rm} models data transfer from memory to registers (*i.e.*, memory load) and \mathbf{mov}_{mr} models the opposite data transfer (*i.e.*, memory store), where $addr$ points to the start location in the memory and mb dedicates the memory size to be manipulated;
- *Control Flow Instructions:* We have modeled the required semantics of instructions that govern the control flow. For instance, \mathbf{jcc} formalizes a control transfer under the condition η , and \mathbf{call} facilitates procedure calls using either a relative address displacement (\mathbf{call}_i) or an absolute address stored in a register (\mathbf{call}_r);
- *System Instructions:* We have formalized the specialized instructions for system-level functions. For instance, \mathbf{rdtsc} models system API to read the time-stamp counter. \mathbf{nop} is used for instruction alignment and timing adjustments without altering the program state.

3.2 Machine State

The program state is denoted as $\mathcal{S} ::= (\mathcal{R}, \mathcal{M})$, comprising pairs of the register state \mathcal{R} and the memory state \mathcal{M} .

Register Map The register state \mathcal{R} is defined as an injective mapping from the processor registers to values,

$$(\text{Regmap}) \quad \mathcal{R} \in \text{preg} \rightarrow \text{Val}$$

where Val represents values of varying bit widths. For instance, when $\mathcal{R}(\text{RAX})$ (or $\llbracket \text{RAX} \rrbracket$) conform to $\text{Vint } i$, it represents the value in EAX , which is the pseudo-register name of the lower 32-bit portion of RAX . The definition of Val also includes a special constructor Vundef denoting undefined value.

$$\text{Val} ::= \text{Vundef} \mid \text{Vbyte } \text{int8} \mid \text{Vshort } \text{int16} \mid \text{Vint } \text{int32} \mid \text{Vlong } \text{int64}$$

Program Counter Unlike assembly-level semantics where pc increments by 1 for non-control flow instructions, binary-level semantics increases pc by the byte size of each instruction, as determined by the decoder_{x64} . We define **nextinstr** function for updating pc to the address of the next instruction based on the input byte size sz . We use $\mathcal{R}(r)$ to denote the value of register r in register map \mathcal{R} , $\mathcal{R}\{r \leftarrow v\}$ for updating it to v , and \mathcal{R}^{+sz} to signify **nextinstr**(\mathcal{R}, sz).

$$\text{nextinstr}(\mathcal{R}, sz) \stackrel{\text{def}}{=} \mathcal{R}\{pc \leftarrow \mathcal{R}(pc) + \text{Vlong } sz\}$$

Memory Our memory model is inspired by CompCert [13]. The memory state \mathcal{M} is defined as a partial mapping from a 64-bit address to a byte.

$$(\text{Memory}) \quad \mathcal{M} \in \text{int64} \rightarrow \text{int8}$$

The memory model provides several fundamental operations, including:

- **load**(mb, \mathcal{M}, va) = $\lfloor v \rfloor$: Read the value v in memory block mb at address va from \mathcal{M} .
- **store**(mb, \mathcal{M}, va, v) = $\lfloor \mathcal{M}' \rfloor$: Store value v into memory block mb at address va , returning the updated memory \mathcal{M}' .

load and **store** may fail in cases of invalid memory access. Therefore, those operations return option types, with $\lfloor v \rfloor$ (*i.e.*, *Some v*) indicating success, and \emptyset (*i.e.*, *None*) indicating failure.

3.3 Semantics

We define the semantics of instructions that interact with registers and memory using the transition $(\mathcal{R}, \mathcal{M}) \xrightarrow{\text{ins}} (\mathcal{R}', \mathcal{M}')$. Typical instructions are selected from each of the four base sets to demonstrate the process of semantic formalization.

Arithmetic and Logic Instructions The 32-bit **addl_{rr}** $r_d r_s$ instruction first verifies that both the source register r_s and destination register r_d contain 32-bit integers, and uses function **bszie** to determine the instruction length n . The value in r_s is then added to r_d , followed by an update to pc .

$$\frac{\text{Vint } v_1 = \mathcal{R}(r_d) \quad \text{Vint } v_2 = \mathcal{R}(r_s) \quad \text{bsize}(\text{addl}_{rr} r_d r_s) = n}{(\mathcal{R}, \mathcal{M}) \xrightarrow{\text{addl}_{rr} r_d r_s} (\mathcal{R}^{+n}\{r_d \leftarrow \text{Vint}(v_1 + v_2)\}, \mathcal{M})} \quad (\text{Add32})$$

Data Transfer Instructions x86-64 adopts the **mov** instructions to implement load and store operations. The function **eval_addr** is to compute the 64-bit address $addr$ for memory access.

$$\frac{\text{eval_addr}(addr, \mathcal{R}) = \text{Vlong } va \quad \text{bsize}(\text{mov}_{rm} r_d addr mb) = n \quad \lfloor v \rfloor = \text{load}(mb, \mathcal{M}, va)}{(\mathcal{R}, \mathcal{M}) \xrightarrow{\text{mov}_{rm} r_d addr mb} (\mathcal{R}^{+n}\{r_d \leftarrow v\}, \mathcal{M})} \quad (\text{Load})$$

$$\frac{\text{eval_addr}(addr, \mathcal{R}) = \text{Vlong } va \quad \text{bsize}(\text{mov}_{mr} r_d addr mb) = n \quad \lfloor \mathcal{M}' \rfloor = \text{store}(mb, \mathcal{M}, va, \mathcal{R}(r_s))}{(\mathcal{R}, \mathcal{M}) \xrightarrow{\text{mov}_{mr} addr r_s mb} (\mathcal{R}^{+n}, \mathcal{M}')} \quad (\text{Store})$$

The **mov_{rm}** $r_d addr mb$ instruction first computes the exact address value va by **eval_addr**, then uses **load** to retrieve the value v in the memory block mb at the address va , loading it into register r_d . Conversely, **mov_{mr}** $addr r_s mb$ uses **store** to transfer the contents of register r_s into mb at va , and returns the new memory \mathcal{M}' . Both of the operations end with an update to pc .

Control Flow Instructions Here we give the semantics of instructions that control execution flow. The **jcc** ηd instruction takes an integer displacement d and conditionally performs a jump. Specifically, if the condition η , as assessed by the Rflags, is satisfied, the jump targets a specific address; otherwise, execution proceeds to the next sequential instruction. **eval_cond** is used to verify whether the status register value $\mathcal{R}(f)$ holds the condition η .

$$\frac{\text{eval_cond}(\eta, \mathcal{R}(f)) = \text{True}}{(\mathcal{R}, \mathcal{M}) \xrightarrow{\text{jcc } \eta d} (\mathcal{R}^{+d}, \mathcal{M})} \quad (\text{Jcc-T})$$

$$\frac{\text{eval_cond}(\eta, \mathcal{R}(f)) = \text{False} \quad \text{bsize}(\text{jcc } \eta d) = n}{(\mathcal{R}, \mathcal{M}) \xrightarrow{\text{jcc } \eta d} (\mathcal{R}^{+n}, \mathcal{M})} \quad (\text{Jcc-F})$$

The **call_r** r and **call_i** i instructions first push the current pc onto the stack. This is, decreasing the stack pointer register RSP by the byte size of pc (always 8 in our case) and storing pc at the location pointed by RSP . Then set pc to the 64-bit target address (va for absolute address or i for relative address).

$$\frac{\mathcal{R}(r) = \text{Vlong } va \quad \mathcal{R}' = \mathcal{R}\{RSP \leftarrow \mathcal{R}(RSP) - 8\} \quad \lfloor \mathcal{M}' \rfloor = \text{store}(M64, \mathcal{M}, \mathcal{R}(pc), \mathcal{R}(RSP))}{(\mathcal{R}, \mathcal{M}) \xrightarrow{\text{call}_r r} (\mathcal{R}'\{pc \leftarrow \text{Vlong } va\}, \mathcal{M}')} \quad (\text{Call-R})$$

$$\frac{\mathcal{R}' = \mathcal{R}\{RSP \leftarrow \mathcal{R}(RSP) - 8\} \lfloor \mathcal{M}' \rfloor = \text{store}(M64, \mathcal{M}, \mathcal{R}(pc), \mathcal{R}(RSP))}{(\mathcal{R}, \mathcal{M}) \xrightarrow{\text{call}_i^i} (\mathcal{R}^{'+i}, \mathcal{M}')} \text{ (Call-I)}$$

System Instructions The Solana JIT employs a select set of system-level instructions, notably the **rdtsc**. This instruction reads the value from the 64-bit processor’s timestamp counter *tsc* as *v* and distributes it across registers *RDX* and *RAX*. *lo_32* and *hi_32* are used to extract the lower and higher 32-bit values from *v*, respectively.

$$\frac{Vlong\ v = \mathcal{R}(tsc)}{(\mathcal{R}, \mathcal{M}) \xrightarrow{\text{rdtsc}} (\mathcal{R}'\{RAX \leftarrow Vint\ lo_32(v), RDX \leftarrow Vint\ hi_32(v), \mathcal{M}\})} \text{ (rdtsc)}$$

4 Modeling x86-64 Encoder and Decoder

This section introduces a formal model of our x86-64 encoder-decoder pair tailored for the Solana JIT. We begin with the relevant x86-64 encoding specification. Then we present the encoder algorithm, which clarifies the process of translating assembly semantics into machine-readable binary codes. Finally, we illustrate the decoder algorithm, which elevates the x86-64 semantics from the binary level back to the assembly.

4.1 x86-64 Encoding Specification

To handle the complexity of CISC-styled instruction formats, we identify 24 fundamental x86-64 encoding patterns, *i.e.*, 4 primary patterns each comprising several subsidiary rules within the Solana JIT implementation. The first primary pattern, *R1*, addresses default operand configurations that commence directly with an opcode byte. Subsequent primary patterns, *R2* through *R4*, incorporate legacy, REX, and escape prefixes. For simplification, we list only the essential subsidiary rules but our formal model covers all the patterns used by Solana JIT.

- *R1*: *ins* that starts with an opcode byte, devoid of any prefix enhancements. Sub-patterns include:
 - *R1.1* [opcode]: *ins* exclusively comprising a singular opcode byte.
 - *R1.2* [opcode + modrm]: *ins* that incorporates both an opcode byte and a ModR/M byte for registry access.
 - *R1.3* [opcode + modrm + imm]: *ins* that integrates an opcode byte, a ModR/M byte, and a 1/2/4/8-byte immediate value.
 - *R1.4* [opcode + displacement]: *ins* that features an opcode byte with a 1/2/4-byte displacement value, used for direct addressing (see [subsection 3.1](#) memory addressing mode 1).
 - *R1.5* [opcode + modrm + displacement]: *ins* that consists of an opcode byte, a ModR/M byte, and a displacement value, to enhance the addressing capabilities (mode 2).

- $R1.6[\text{opcode} + \text{modrm} + \text{sib} + \text{displacement}]$: *ins* that includes an opcode byte, a ModR/M byte, a SIB byte and a displacement value, applicable to enable complex addressing (mode 3).
- $R2$: *ins* with a REX prefix that enhances operand interactions by extending the default 32-bit operand size to 64 bits when REX.W is set, and by enabling access to the extended registers (R8-R15) through proper setting of REX.R, REX.X, and REX.B. Sub-patterns include:
 - $R2.1[\text{rex} + \text{opcode}]$
 - $R2.2[\text{rex} + \text{opcode} + \text{modrm}]$
 - $R2.3[\text{rex} + \text{opcode} + \text{modrm} + \text{imm}]$
 - $R2.4[\text{rex} + \text{opcode} + \text{modrm} + \text{displacement} + \text{imm}]$
 - $R2.5[\text{rex} + \text{opcode} + \text{modrm} + \text{sib} + \text{displacement} + \text{imm}]$
- $R3$: *ins* that operates on 16-bit operands requires the legacy prefix (0x66). Sub-patterns include:
 - $R3.1[\text{legacy} + \text{opcode} + \text{modrm} + \text{imm}]$
 - $R3.2[\text{legacy} + \text{opcode} + \text{modrm} + \text{displacement}]$
 - $R3.3[\text{legacy} + \text{rex} + \text{opcode} + \text{modrm} + \text{imm}]$
 - $R3.4[\text{legacy} + \text{rex} + \text{opcode} + \text{modrm} + \text{displacement}]$
- $R4$: Special *ins* that requires an opcode escape sequence prefix (0x0f) applied before the primary opcode. Sub-patterns include:
 - $R4.1[\text{escape} + \text{opcode}]$
 - $R4.2[\text{escape} + \text{opcode} + \text{modrm}]$
 - $R4.3[\text{rex} + \text{escape} + \text{opcode}]$
 - $R4.4[\text{rex} + \text{escape} + \text{opcode} + \text{modrm}]$

4.2 x86-64 Encoder

We define the x86-64 encoder [Algorithm 1](#) following the previous specification. The algorithm processes each assembly instruction *ins* by first identifying and matching it to the corresponding instruction pattern([line 2](#)). Upon successful pattern matching, the algorithm then converts *ins* into its binary representation, effectively encoding the assembly instruction into the target machine code.

The [Algorithm 1](#) involves two important functions: The `ConstructREX` function([line 4](#), [line 11](#)) compiles the W, R, X, and B bits into a byte for the REX prefix. REX is included only if the lower 4-bit resultant value is non-zero; otherwise, it is omitted([line 6](#), [line 14](#)). Similarly, the `ConstructModRM` function([line 5](#), [line 12](#)) combines the 2-bit `mod`, 3-bit `reg`, and 3-bit `r/m` fields into the ModR/M byte. For example, `addlrr rd rs` can match either $R1.2$ ([line 7](#)) or $R2.2$ ([line 9](#)), depending on the registers involved. The R and B bits in the REX byte are set to 1 if r_s and r_d access extended registers, respectively. The W bit is set to 0 for 32-bit operands, and the X bit remains 0, indicating that no SIB code is required for memory access. Finally, with the opcode 0x01 directly referenced in the manual, all the components are pipelined into a binary instruction.

The `addwri rd imm` instruction([line 10](#)) matches either pattern $R3.1$ ([line 15](#)) or $R3.3$ ([line 17](#)), and is prefixed with 0x66 to signify a 16-bit operand size. We employ `u8_list_of_u16` function to convert a 16-bit integer into a list of

Algorithm 1: x86-64 encoder algorithm

Data: ($ins : instruction$)
Result: ($bin : x64_binary$)

```

1  $encoder_{x64} :$ 
2 switch  $ins$  do
3   case  $addl_{rr}, r_d, r_s$  do
4      $rex \leftarrow \text{ConstructREX}(0, 8 \leq s, 0, 8 \leq d);$ 
5      $modrm \leftarrow \text{ConstructModRM}(0b11, s, d);$ 
6     if  $rex = 0x40$  then
7       return  $[0x01, modrm];$  /* R1.2 */
8     else
9       return  $[rex, 0x01, modrm];$  /* R2.2 */
10  case  $addw_{ri}, r_d, imm$  do
11     $rex \leftarrow \text{ConstructREX}(0, 0, 0, 8 \leq d);$ 
12     $modrm \leftarrow \text{ConstructModRM}(0b11, 0b000, d);$ 
13     $l_{imm} \leftarrow \text{u8\_list\_of\_u16}(imm)$ 
14    if  $rex = 0x40$  then
15      return  $[0x66, 0x81, modrm] @ l_{imm};$  /* R3.1 */
16    else
17      return  $[0x66, rex, 0x81, modrm] @ l_{imm};$  /* R3.3 */
18  case  $nop$  do
19    return  $[0x90];$  /* R1.1 */
20  case ... do
21    ...; /* handle other ins */

```

bytes(line 13) and use the ‘@’ operator to concatenate these lists(line 15, line 17). The **nop**(line 18) instruction matches *R1.1*, for which the opcode 0x90 is directly returned(line 19). Other cases follow a similar pattern-matching process but are skipped for simplification.

Although a single assembly instruction may correspond to multiple binary encodings, our encoder is injective (see Lemma 1). This means that while the potential encodings vary based on the information specified in the assembly instruction, only one specific binary representation is generated for each instruction once the information is determined.

4.3 x86-64 Decoder

The decoder Algorithm 2 tackles a tougher challenge to map binary encodings back to the assembly forms. Similarly, our decoder is also injective. That is, for each ins encoded into a sequence of bytes bin , our decoder will faithfully decode bin back to its original instruction. The decoder processes a list of x86-64 binary instructions l_{bin} and uses pc to mark the start of the current instruction. It outputs a corresponding assembly instruction and calculates the byte size of the instruction. This calculated size is essential for correctly branching to the subsequent instruction during the assembly execution.

Algorithm 2: x86-64 decoder algorithm

Data: $(pc : nat), (l_{bin} : x64_binary)$
Result: $(asm : (nat \times instruction) option)$

```

1  $decoder_{x64} :$ 
2  $h0 \leftarrow l_{bin}[pc];$ 
3 if  $h0 = 0x90$  then
4   return  $[(1, \text{nop})]$  ; /* R1.1 */
5 else if  $h0 = 0x66$  then
6    $h1 \leftarrow l_{bin}[pc + 1];$ 
7   if  $h1$  is not rex then
8      $(mod, r_s, r_d) \leftarrow l_{bin}[pc + 2];$ 
9     if  $h1 = 0x81$  then
10       $imm \leftarrow u16\_of\_u8\_list(l_{bin}[pc + 3], l_{bin}[pc + 4]);$ 
11      if  $mod = 0b11$  and  $s = 0b000$  then
12        return  $[(5, \text{addw}_{ri} \ r_d \ imm)]$  ; /* R3.1 */
13      else
14        return  $\emptyset$ ;
15      else
16        ...; /* R3.2 */
17    else
18       $(\_, w, r, x, b) \leftarrow h1;$ 
19      ...; /* R3.3, R3.4 */
20 else if  $h0 = 0x0f$  then
21   ...; /* R4.1, R4.2 */
22 else if  $h0$  is not rex then
23    $(mod, r_s, r_d) \leftarrow l_{bin}[pc + 1];$ 
24   if  $h0 = 0x01$  then
25     if  $mod = 0b11$  then
26       return  $[(2, \text{addl}_{rr} \ r_d \ r_s)]$  ; /* R1.2 */
27     else
28       return  $\emptyset$ ;
29   else
30     ...; /* R1.3, R1.4, R1.5, R1.6 */
31 else
32   ...; /* R2.1, R2.2, R2.3, R2.4, R2.5, R4.3, R4.4 */

```

The algorithm begins by examining the byte ($h0$) at position pc in l_{bin} (line 2). Based on the encoding rules (subsection 4.1), there are several possible outcomes:

- If $h0$ is a single opcode byte for a no-operand instruction, it matches pattern *R1.1*. For example, if $h0$ equals $0x90$ (line 3), the decoder returns the **nop** assembly with a byte size of 1 (line 4).
- If $h0$ is the legacy prefix $0x66$ (line 5), it matches patterns *R3.1* to *R3.4*. Analysis of the next byte $h1$ follows (line 6):
 - If $h1$ is not a REX prefix (line 7), it matches *R3.1* or *R3.2*. In this case, $h1$ serves as the opcode, and the following byte is the ModR/M byte.

Using a mapping, the decoder extracts the `mod`, `reg`, and `r/m` (*i.e.*, r_d) fields(line 8), requiring further bytes to fully determine the instruction. For instance, if $h1$ is 0x81(line 9) followed by a 16-bit immediate imm , and `mod` is 0b11 with `reg` at 0b000(line 11), the instruction is confirmed as `addwri rd imm`(line 12) with length of 5. `u16_of_u8_list` function is to convert two bytes into a 16-bit integer(line 10). In this case, `reg` field extends the opcode rather than serving as a direct operand.

- If $h1$ is a REX prefix, the algorithm seeks to align with $R4.3$ and $R4.4$. It extracts the REX.WRXB fields from $h1$ (line 18) and then proceeds with an analysis akin to the previously described, which will not be reiterated.
- If $h0$ is the opcode escape prefix 0x0f, it matches $R4.1$ to $R4.4$, and the decoder should disassemble an instruction with an escape sequence(line 21).
- For other values of $h0$, the algorithm checks for a REX prefix(line 22):
 - If $h0$ is not a REX prefix, the decoder matches it with $R1.2$ to $R1.6$ to identify 32-bit instructions without extended registers, or certain 8-bit instructions. An example is provided with the parsing of the `addlrr rd rs` instruction(line 24) with length of 2(line 26).
 - If $h0$ is a REX prefix, the decoder matches it with $R2.1$ to $R2.5$, $R4.3$ or $R4.4$ to identify either 8/32-bit instructions that access extended registers or 64-bit instructions(line 32).

In instances where none of the defined instruction patterns are met, the decoding is deemed invalid and the algorithm will return \emptyset (line 14, line 28) because the Solana JIT does not generate this pattern.

5 Verifying Encoder-Decoder Equivalence

This section demonstrates that the formal model of our x86-64 encoder-decoder pair satisfies the equivalence property. The equivalence property confirms that the bidirectional mapping between assembly and binary instructions allows seamless conversion between the two without any information loss. This property is essential to verify the correctness of the entire Solana eBPF JIT compiler workflow, as shown in Figure 1.

To prove equivalence, we first discuss two lemmas:

- *Encoder implies decoder*(Lemma 1): given that a list of x86-64 assembly instructions is encoded into binary form, prove that the encoded instructions can always be decoded back to the original assembly code.
- *Decoder implies encoder*(Lemma 2): given that a list of binary instructions is decoded into assembly form, prove that the decoded instructions can always be re-encoded to reproduce the original binary code.

We use $l \sqsubseteq_n L$ to represent a list l as a contiguous sublist within a larger list L starting at position n . Here, $l[i]$ refers to the i -th element of l .

$$l \sqsubseteq_n L \stackrel{\text{def}}{=} \forall i \ v. \ l[i] = \lfloor v \rfloor \Rightarrow L[n+i] = \lfloor v \rfloor$$

In the proof context, we declare $l_{bin} \sqsubseteq_n L_{bin}$ as an assumption to reflect that each jited x86-64 binary sequence l_{bin} of Solana JIT rule is a sublist within the complete jited x86-64 binary code L_{bin} of the entire Solana JIT compiler.

Lemma 1 (x86-64 Encoder $_Implies_Decoder$). *If $l_{bin} \sqsubseteq_{pc} L_{bin}$ and $encoder_{x64}(ins) = l_{bin}$, then $decoder_{x64}(pc, L_{bin}) = \lfloor (length(l_{bin}), ins) \rfloor$*

Proof. The proof begins by conducting a case analysis on the assembly instruction ins , where each instance generates a subgoal. We conduct further case analysis on each of its register-related operands and rely on the high degree of proof automation of Isabelle/HOL to solve all subgoals.

Lemma 2 (x86-64 Decoder $_Implies_Encoder$). *If $l_{bin} \sqsubseteq_{pc} L_{bin}$ and $decoder_{x64}(pc, L_{bin}) = \lfloor (length(l_{bin}), ins) \rfloor$, then $encoder_{x64}(ins) = l_{bin}$*

Proof. The proof also begins by case analysis on ins rather than on L_{bin} to maintain a more coherent and focused proof structure and reduce complexity. For each ins and its operands, relevant function definitions are unfolded to enable detailed examination at the bit level. We then conduct case analysis on l_{bin} to extract each byte and try to match it with ins . Simplification of the proof is achieved using the existing lemmas on bit operators provided by Isabelle/HOL.

Theorem 1 (Encoder-Decoder Equivalence). *If $l_{bin} \sqsubseteq_{pc} L_{bin}$, then $decoder_{x64}(pc, L_{bin}) = \lfloor (length(l_{bin}), ins) \rfloor \iff encoder_{x64}(ins) = l_{bin}$*

Proof. The proof is established directly by applying [Lemma 1](#) and [Lemma 2](#).

Status of the Proof. At the time of writing, the x86-64 binary model (including semantics, encoder, and decoder) and the proofs of [Lemma 1](#) are done, sufficiently establishing the groundwork for proving the main theorem of Solana JIT correctness ([subsection 2.1](#)). While the mechanization of [Lemma 2](#) continues due to the complex bit operations, completion of the full proof is anticipated soon.

6 Related Work

There have been many projects that formalize the semantics of various ISAs either as their main contribution or as part of their infrastructure. Pioneering work include research on ISAs such as x86 ([\[15,8\]](#), using \mathbb{K} [\[4\]](#)), ARM [\[6,29\]](#), RISC-V [\[19\]](#), SPARC [\[9,10,24\]](#), as well as verified Compilers *e.g.*, CompCert, CakeML [\[11\]](#). We gain insights from these studies to inform our own research. This section mainly discusses related work on the formalization of x86 ISA and the verification of encoders and decoders. We compare our x86-64 formal model with existing studies and highlight our primary contributions to the verification of the Solana JIT.

Formalization of the x86 ISA. Various approaches have been taken to formalize the x86 ISA, but many do not align with our objectives. Our main goal is to provide the infrastructure that supports the verification of the Solana JIT correctness, which demands the x86-64 binary semantics and the equivalence proof of the encoder-decoder pair.

Sail is a language semantics framework tailored for describing ISA semantics. Sail x86-64 semantics is initially translated from an ACL2 formal model by Goel *et al.* [7] and then re-translated into Isabelle/HOL scripts. However, Sail significantly complicates the task of proving encoder-decoder equivalence due to its formalization of x86 semantics at the bit level. We shift to the byte level to avoid operations and transformations on individual bits.

Dasgupta *et al.* [4] employ \mathbb{K} framework to define formal semantics for over 3000 user-level x86-64 instructions of Intel Haswell processor. This is so far the most comprehensive formalization of x86-64 ISA to our knowledge. However, it does not include specific mechanisms for instruction encoding and decoding.

Verbeek *et al.* [23] provide semantics for basic x86-64 assembly instructions in Isabelle/HOL, including arithmetic operations, jumps, and others. But this semantics is more limited than ours, covering only 120 instructions. It also maintains semantics at the assembly level and lacks mechanisms for binary analysis.

CakeML is a fully verified functional programming language compiler proven in HOL4 [20]. It supports verified end-to-end compilation into target concrete machine code for realistic x86-64 architecture, but the target proofs focus on the consistency between the assembly bytes in the memory and the binary sequences. Therefore, to our knowledge, there is no proof of encoder-decoder equivalence in the CakeML correctness theorem.

CompCert is a fully verified compiler for the C programming language in Coq. Its verification covers compilation passes beginning with the CompCert C AST, and extending to various target assembly languages (*e.g.*, x86-64, x84, ARM, RISC-V, etc). The scope of verification still does not extend to the binary level. Nevertheless, subsequent projects based on CompCert have yielded significant insights (*e.g.*, FM-JIT [2], and CompCertELF [26,27]).

Verification of Encoders and Decoders. To delve into binary-level analysis of the x86 ISA, various projects incorporate encoders and decoders, ensuring their correctness through formal verification.

CompCertELF extends CompCert to support verified separate compilation from C programs to the Executable and Linkable Format (ELF). CompCertELF develops a verified encoder-decoder pair as a key component to link the CompCert assembly code with 32-bit x86 binary code in ELF format. Our model diverges distinctively from CompCertELF in several parts, serving different purposes. Firstly, it only supports the 32-bit x86 backend, whereas our binary model is built on the 64-bit x86 architecture and accommodates backward bit modes. Secondly, the encoder-decoder pair in CompCertELF doesn't satisfy the equivalence property, meaning that applying encoding and then decoding to instructions does not accurately reproduce the original assembly inputs.

Rocksalt [15] is another pioneering work that explores the x86 ISA at the binary level. It introduces a formally verified checker for Google’s Native Client in Coq, targeting the 32-bit x86 processor. However, Rocksalt fails to develop a deterministic decoder since it lacks an encoder to prove consistency. Moreover, the subset of instructions it supports is relatively smaller compared to ours.

There are also some projects that focus on BPF JITs verification, which align more closely with our long-term goal of verifying the Solana JIT compiler. A common practice for them is also to employ encoders and decoders as part of the JIT correctness proof. However, these efforts are currently insufficient to fully support the Solana JIT proof. For example, Jitk [25] extends the CompCert to translate classic BPF to assembly for building verified in-kernel interpreters. Jitterbug [16] offers a framework for generating formal specifications to aid JIT correctness proofs, and has been applied to several real-world BPF JITs in Linux. Compared to our work, Jitterbug incurs a larger TCB by relying on external SMT solvers for symbolic execution. CertBPF-JIT [28] provides a verified JIT compiler for RIOT-OS eBPF virtual machine in Coq, but it is currently limited to the ARM architecture and supports a small subset of arithmetic instructions which is too restricted for our application in the Solana JIT proof.

7 Conclusion and Future work

In this paper, we present the infrastructure to verify Solana JIT correctness by formalizing a necessary subset of x86-64 binary semantics and demonstrating the equivalence proof of the x86-64 encoder-decoder pair. We first lift the target x86-64 binaries to the higher-level assembly representation, where we define the syntax, machine state, and semantics for all x86-64 instructions employed by the Solana JIT. We then tackle the complexities of low-level x86-64 ISA semantics by modeling the encoder and decoder specific to the Solana JIT. Our model adheres to the defined x86-64 encoding specifications for precise transition between the assembly and the binary level. Additionally, our proof of the encoder-decoder equivalence significantly simplifies the main theorem of the Solana JIT correctness, allowing further verification to be conducted at the assembly level.

Our future work will focus on a complete formal verification of the Solana x86-64 eBPF JIT compiler. While this study has established the foundational binary semantics for the x86-64 decoder correctness in Solana JIT, further research is going to explore additional facets, including the eBPF binary decoder and the JIT rules. Additionally, we are continuing to validate our binary semantics. We leverage the extraction mechanism Isabelle/HOL provides to automatically generate executable semantics in OCaml. These semantics are then validated against the official x86-64 test suite to confirm that our binary semantic model supports real-world programs.

Acknowledgements This work has been supported by the Natural Science Foundation of China (Grant No. 62132014 and No. U2341212), and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security.

References

1. Armstrong, A., Krishnaswami, N., Sewell, P., Wassell, M.: Formalisation of minisail in the isabelle theorem prover. In: Proceedings of the Automated Reasoning Workshop (2018)
2. Barrière, A., Blazy, S., Pichardie, D.: Formally verified native code generation in an effectful jit: turning the compcert backend into a formally verified jit compiler. *Proceedings of the ACM on Programming Languages* **7**(POPL), 249–277 (2023)
3. Bertot, Y., Castran, P.: *Interactive Theorem Proving and Program Development: Coq’Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edn. (2010)
4. Dasgupta, S., Park, D., Kasampalis, T., Adve, V.S., Roşu, G.: 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. pp. 1133–1148 (2019)
5. Fleming, M.: *A Thorough Introduction to eBPF* (2017)
6. Fromherz, A., Giannarakis, N., Hawblitzel, C., Parno, B., Rastogi, A., Swamy, N.: A verified, efficient embedding of a verifiable assembly language. In: *Principles of Programming Languages (POPL 2019)*. ACM (January 2019), <https://www.microsoft.com/en-us/research/publication/a-verified-efficient-embedding-of-a-verifiable-assembly-language/>
7. Goel, S.: Formal verification of application and system programs based on a validated x86 ISA model. Ph.D. thesis, University of Texas at Austin (2016)
8. Goel, S., Slobodová, A., Sumners, R., Swords, S.: Verifying x86 instruction implementations. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. pp. 47–60 (2020)
9. Hou, Z., Sanan, D., Tiu, A., Liu, Y.: A formal model for the sparcv8 isa and a proof of non-interference for the leon3 processor. *Archive of Formal Proofs* (October 2016), <https://isa-afp.org/entries/SPARCV8.html>, Formal proof development
10. Hóu, Z., Sanan, D., Tiu, A., Liu, Y., Hoa, K.C., Dong, J.S.: An isabelle/hol formalisation of the sparc instruction set architecture and the tso memory model. *Journal of Automated Reasoning* **65**, 569–598 (2021)
11. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: Cakeml: a verified implementation of ml. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. p. 179–191. POPL ’14, Association for Computing Machinery, New York, NY, USA (2014). <https://doi.org/10.1145/2535838.2535841>
12. Leroy, X.: Formal verification of a realistic compiler. *Communications of the ACM* **52**(7), 107–115 (2009), <http://xavierleroy.org/publi/compcert-CACM.pdf>
13. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA (Jun 2012), <https://inria.hal.science/hal-00703441>
14. Lu, J., Yuan, S., Sanan, D., Zhao, Y.: Solana x64 Semantics (2024), <https://github.com/shenghaoyuan/Solana-x64-Semantics>
15. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.B., Gan, E.: Rocksalt: better, faster, stronger sfi for the x86. In: *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. pp. 395–404 (2012)
16. Nelson, L., Geffen, J.V., Torlak, E., Wang, X.: 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). pp. 41–61. USENIX Association (Nov 2020), <https://www.usenix.org/conference/osdi20/presentation/nelson>
17. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: a proof assistant for higher-order logic. Springer-Verlag, Berlin, Heidelberg (2002)
 18. Ray, S.: Introduction to ACL2, pp. 25–49. Springer US, Boston, MA (2010). https://doi.org/10.1007/978-1-4419-5998-0_3, https://doi.org/10.1007/978-1-4419-5998-0_3
 19. RISC-V Coq Team: RISC-V Specification in Coq (2024), <https://github.com/mit-plv/riscv-coq>
 20. Slind, K., Norrish, M.: A brief overview of hol4. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) Theorem Proving in Higher Order Logics. pp. 28–32. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
 21. Solana-labs: Solana rBPF (2024), <https://github.com/solana-labs/rbpf>
 22. Tyma, P.: Why are we using java again? Commun. ACM **41**(6), 38–42 (jun 1998). <https://doi.org/10.1145/276609.276617>, <https://doi.org/10.1145/276609.276617>
 23. Verbeek, F., Bharadwaj, A., Bockenek, J., Roessle, I., Weerwag, T., Ravindran, B.: X86 instruction semantics and basic block symbolic execution. Archive of Formal Proofs (October 2021), https://isa-afp.org/entries/X86_Semantics.html, Formal proof development
 24. Wang, J., Fu, M., Qiao, L., Feng, X.: Formalizing sparcv8 instruction set architecture in coq. Sci. Comput. Program. **187**, 102371 (2020). <https://doi.org/10.1016/J.SCICO.2019.102371>, <https://doi.org/10.1016/j.scico.2019.102371>
 25. Wang, X., Lazar, D., Zeldovich, N., Chlipala, A., Tatlock, Z.: Jitk: A trustworthy In-Kernel interpreter infrastructure. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). pp. 33–47. USENIX Association, Broomfield, CO (Oct 2014), https://www.usenix.org/conference/osdi14/technical-sessions/presentation/wang_xi
 26. Wang, Y., Xu, X., Wilke, P., Shao, Z.: Compcertelf: verified separate compilation of c programs into elf object files. Proceedings of the ACM on Programming Languages **4**(OOPSLA), 1–28 (2020)
 27. Xu, X., Wu, J., Wang, Y., Yin, Z., Li, P.: Automatic generation and validation of instruction encoders and decoders. In: Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33. pp. 728–751. Springer (2021)
 28. Yuan, S., Besson, F., Talpin, J.P.: End-to-end mechanized proof of a jit-accelerated ebpf virtual machine for iot. In: Computer Aided Verification. Springer International Publishing, Cham (2024)
 29. Yuan, S., Talpin, J.P.: Verified functional programming of an iot operating system’s bootloader. In: Proceedings of the 19th ACM-IEEE International Conference on Formal Methods and Models for System Design. p. 89–97. MEMOCODE ’21, Association for Computing Machinery, New York, NY, USA (2021). <https://doi.org/10.1145/3487212.3487347>