Auditing a verified, AI-generated compiler by formalizing its context

Leo Alt
powdr

Arie Gurfinkel
Waterloo, Certora

Gereon Kremer
Independent

Alex Ozdemir
Georgia Tech, Certora
aozdemir@gatech.edu

Christian Reitwiessner
powdr

Georg Wiese
powdr

Abstract

Verified code generation is a promising way to build trustworthy software. Humans settle on a specification, and an AI writes both the implementation and a machine-checked proof that the implementation meets it. Thus, all remaining risk lies in the specification, and the specification must be audited. That audit is easy when the specification is small and self-contained. However, most software is one module of a larger system, and a module's specification may be neither small nor self-contained: it depends on what the rest of the system guarantees. We report on our audit of such a module: the autoprecompile optimizer of powdr, a compiler for zero-knowledge virtual machines (zkVMs). It had been ported to Lean by AI and proven correct against a human-reviewed specification. That specification was stated over a single chip, yet its correctness rested on unverified assumptions about the rest of the zkVM. We first tried to audit those assumptions manually, through reading and discussion, and we failed. So, instead, we applied *more* formalization, in three ways. First, the auditors *edited* the specification, rather than arguing about it. Second, we wrote a *meta-specification*, which formalizes the module's context and derives the specification from it. Third, we *dynamically validated* the assumptions that survived, on real inputs, using decidable static analyses. We describe each methodology and how it improved the assurance case for our codebase.

1 Introduction

Together, theorem proving and AI are great tools for engineering trustworthy software. Morrison [19] recently documented one example: a high-performance `zip` implementation in Lean [20]. First, Morrison wrote a specification for the `zip` compressed file format. Then, an AI wrote the implementation and a proof that it meets the specification. The result was both verified and fast—faster even than the pure-Rust `miniz_oxide` [5].

This workflow is called *verified code generation*: humans write the specification, and AI writes the implementation and a proof connecting them. It is attractive because humans write specifications instead of code and proofs.

This worked for `zip` for two reasons. First, the specification is short, classical, and self-contained. Second, one person owned the entire software artifact, so a wrong or slow implementation could be discarded and regenerated, since nothing outside depended on its internals.

However, most software has neither property. A large system is a collection of modules with different owners, and a module's specification is rarely self-contained: it refers to what the module's environment provides, and to what the module must maintain on the environment's behalf. So, even when the

NeurIPS 2026 Workshop on AI for Verifiable Coding.

whole system has a small specification, a single module's can be large, and wrong in ways that only the rest of the system reveals. This raises two questions for verified code generation in a modular setting. First, how does a module's owner learn its specification? Second, how can a specification separate concerns, so that the module can be audited on its own?

Our setting We encountered these questions when auditing a concrete verified-code-generation project. powdr [28] is an optimizing compiler that generates *autoprecompiles* (APCs) for a zero-knowledge virtual machine (zkVM). To make an APC, powdr fuses the instructions of a hot basic block into one large chip, and then optimizes it. The optimizer produces the savings, and it also carries the correctness risk (§2). powdr's optimizer had been ported from Rust to Lean with Claude Code ("the AI" below), and proven correct against a specification that its authors had directed and reviewed [25]. The result was roughly 50k lines of implementation and proof, behind a 450-line specification. We set out to audit that specification. Our goal was to strengthen the assurance case for the optimizer, not to assess it independently, so the audit was a collaboration between the optimizer's developers and its auditors.

At a high level, the specification was natural. It said that the optimized chip C' and the original chip C are *equi-effectful*: every satisfying assignment of one has a counterpart in the other with the same effect on the rest of the machine. However, the details were more concerning, because the specification made two assumptions. First, it simplified the satisfiability semantics for chips. Second, it applied only to assignments respecting a predicate $\mathsf{Admissible}(a)$. Each was informally justified by an appeal to other parts of the VM. None of these arguments was formal, and none could be, because the specification did not model the rest of the VM.

Our audit This paper is an experience report on auditing that specification. Our first attempt was to read it (§3). After several days, we had found no clear bug, but we were also not convinced that the assumptions were sound. So, to make progress, we tried three different methodologies, which this paper aims to explain. First, we *edited* the specification (§4). Rather than argue about whether an assumption is acceptable, we changed it into one that we all found more acceptable. This is newly practical, because AI can make the change without codebase expertise, and Lean identifies exactly which proofs the change breaks. However, this approach has no clear stopping criterion. Second, we wrote a *meta-specification* (§5). That is, we formalized the VM around the optimizer, specified (but did not verify) its other modules, and proved that the original chip-level specification implies VM-level equivalence. This makes the chip-level specification untrusted: it is absorbed into the implementation as an intermediate abstraction. Third, we *dynamically validated* what was left (§6): a set of per-chip assumptions $\mathsf{Legal}(C)$. Here, we had AI write sound (but not complete) decidable analyses for Legal, and then we ran them on real APCs.

2 Background: zkVMs and autoprecompiles (APCs)

We give the minimum needed to understand the powdr optimizer that we audited. For more details, see the powdr documentation [28] and the paper on the OpenVM zkVM that it optimizes [21].

Chips and buses A zkVM proves that a public assembly program, run on private data, produced a claimed result. It is built from *chips*—also called *circuits*—each defined over a finite field $\mathbb{F}$ of prime order p by (i) a set of symbolic algebraic constraints, and (ii) a list of symbolic *bus interactions*. A bus interaction has a bus ID, a message $m \in \mathbb{F}^*$ (a fixed-length vector), and a multiplicity $\mu \in \mathbb{F}$. Its effect is the point function sending m to μ and every other input to 0. For an assignment a to C's variables, let $\mathsf{Sat}(C, a)$ hold when a satisfies C's algebraic constraints, and let $\mathsf{Effect}(C, a)$ be the sum of C's interaction effects, as functions.

A VM assignment $\mathbf{a}$ gives, for each chip C_i, a list of assignments $a_{i,1}, \ldots, a_{i,n_i}$—one per *instance* of that chip, e.g. one per execution of the instruction it implements. It is satisfying when

$$\forall i, j.\ \mathsf{Sat}(C_i, a_{i,j}) \qquad \text{and} \qquad \textstyle\sum_{i,j} \mathsf{Effect}(C_i, a_{i,j}) = 0, \tag{1}$$

where the right-hand 0 is the zero function, and when the number of instances of each chip is less than some fixed limit. OpenVM's proof system enforces both conditions, so they are our ground truth. The fact that the effects sum to zero is called *balancing*.

The machine on top of the buses Bus IDs come from a fixed set, with each bus's messages encoding different types of data, by convention. The *execution bridge* carries $(\mathrm{pc}, \mathrm{time})$ pairs—program counter and time—passed from one instruction to the next. The *memory* bus carries $(\mathrm{address}, \mathrm{value}, \mathrm{time})$ records, in the style of Blum et al. [2]. The zkVM's registers are just an address range in memory. Each lookup table gets its own bus, which carries values in the table.

Chips split into two kinds: *system* chips and *instruction* chips. The system chips are fixed machinery: memory initialization and finalization, program start and end, the input (hint) chip, and one sink chip for each lookup table. The instruction chips implement the VM's instructions, including the APCs, each of which implements a whole block of instructions.

Several *conventions* make this into a machine, and none is checked by any single chip. We give four examples. First, each instruction chip receives exactly one $(\mathrm{pc}, \mathrm{time})$ pair and sends exactly one, so that "time" increases during execution. Second, each instruction chip's multiplicities lie in $\{-1, 0, +1\}$, meaning "receive", "no interaction", and "send". Third, each table chip is the only chip that receives on its bus, and only receives in-table messages, so that balancing forces every send to be in the table. Last, only bytes are sent on the memory bus. These conventions are required of all instruction chips, but some system chips are exempt. Together with the behavior of the system chips, they form a global inductive argument: each chip may assume that all others follow the conventions, and must follow them itself. Thus, the conventions hold globally.

Autoprecompiles Every instruction pays for its interactions, whether it uses them or not: an opcode lookup, two bridge interactions, and memory interactions for its register reads and writes. These interactions are essential in general, because they are the VM's only form of dynamic dataflow, and each kind of dataflow is essential for some instruction. Conditional jumps pass the outgoing timestamp to a data-dependent successor, which is what the bridge interactions provide. RAM accesses receive data from a data-dependent prior instruction, which is what the memory interactions provide. Each instruction type must only apply when the program counter points to its opcode, which is what the opcode lookup provides. However, inside a basic block of a fixed program, the pcs, opcodes, and register dataflow are all fixed, so these interactions are overhead.

The powdr optimizer eliminates this overhead. It copies all of a hot basic block's instruction chips into one large chip, with the pcs, opcodes, and timestamp changes pinned. The result is an APC, which powdr then *optimizes*: it removes matching bus interactions (equal messages m, opposite multiplicities μ), drops redundant lookups and algebraic constraints, and eliminates variables. This commonly reduces costs by 2.5–6× [11]. The optimizer is what we audited. It supports two zkVMs, OpenVM [21] and SP1 [30], but our audit focused on its correctness for OpenVM.

A tiny example Figure 1 shows a block of two instructions, at pcs π and $\pi + 4$: `addi x3, x1, 5`, then `add x4, x3, x2`. Copying both chips into one APC pins the pcs and timestamps, which makes two pairs of interactions match exactly. The first instruction sends the $(\mathrm{pc}, \mathrm{time})$ pair that the second receives, and writes the `x3` record that the second reads. Each pair has equal messages and opposite multiplicities, so its effects sum to the zero function and the optimizer deletes it. What survives is the block's outer bridge pair: the APC receives once, at entry, and sends once, at exit.

Partial concatenation

From	Bus	μ	Message m	
1	bridge	-1	$(\pi,\ t_0)$	
1	bridge	$+1$	$(\pi + 4,\ t_1)$	†
1	memory	$+1$	$(\texttt{x3},\ a + 5,\ t_1)$	‡
2	bridge	-1	$(\pi + 4,\ t_1)$	†
2	memory	-1	$(\texttt{x3},\ a + 5,\ t_1)$	‡
2	bridge	$+1$	$(\pi + 8,\ t_2)$	

After optimization

Bus	μ	Message m
bridge	-1	$(\pi,\ t_0)$
bridge	$+1$	$(\pi + 8,\ t_2)$

Figure 1: Optimization of the two-instruction block `addi x3, x1, 5`; `add x4, x3, x2`, where a is the value in `x1`. We list only the relevant interactions: each instruction also has an opcode lookup and memory interactions for `x1`, `x2`, and `x4`, none of which cancel. The rows marked † and ‡ are the two matching pairs—equal messages, opposite multiplicities—so each pair's effects sum to the zero function and both rows are deleted.

Our starting point Earlier work formulated a correctness definition for this optimizer: *equi-effectfulness*. C' is a *sound* replacement for C when every satisfying assignment of C' has a counterpart in C with equal effects, and *complete* when the reverse holds; correctness requires both. That work did translation validation [23] for this definition on concrete C and C', with an SMT solver [11]. Then, the powdr team ported the optimizer to Lean with AI, to prove it correct against a specification. Their specification seemed plausible, and our task was to check it precisely.

3 Attempting a manual audit

Our first approach was a traditional (manual) audit: we read the specification and discussed it. This was challenging because the specification did not state its equi-effectfulness claim outright. Instead, it qualified the claim in two ways. First, satisfiability was not the Sat of Eq. (1), but a more restrictive definition. Second, completeness applied only to *some* assignments: those satisfying an admissibility predicate $\mathsf{Admissible}(a)$. Each qualification rested on further assumptions, justified informally by the behavior of other chips in the VM. We found no clear bug in any assumption, but we were *also* unsure that the zkVM's design really implied any of them. We give three examples below.

Assumed lookup semantics As noted, the specification's satisfiability semantics were narrower than Sat (1). For example, they constrained lookup interactions to be in the set that their lookup bus was designed for. This was concerning because the cryptography does not check that predicate directly. Instead, a table chip receives in-set values on the bus, and the VM assumes that every other chip only sends on it; together, the table chip, balancing, and the conventions of §2 ensure that all sent values are in the table. However, balancing is an equation modulo p, so that argument breaks if there are p sends on the bus. Preventing this requires a bound, and that bound was implicit in the narrower definition of satisfiability. Whether it holds is a question about the whole VM.

Order-sensitive admissibility The completeness specification applied only to assignments a satisfying $\mathsf{Admissible}(a)$, which required that after a send to a memory address, the "next" receive from that address—with no intervening interaction there—carries the same data. "Next" was defined syntactically, as the next *interaction in the chip's interaction list*. However, the VM's semantics, Eq. (1), is a sum over interactions, so it cannot see that order. So for a chip that happens to list some send before its receive, every assignment is inadmissible, and the optimizer is free to lose that assignment. It was not clear whether that loss was acceptable.

Implicit bounds on timestamps Last, many parts of the specification dealt with VM timestamps, which, like all VM data, are field elements, wrapping modulo p. So, inductive arguments over time require overflow to be ruled out, which requires a bound on the number of instructions. That bound was implicit, hidden behind the assumptions above about syntactic order, and it was not clear whether the VM met it, or under which conditions.

Conclusion Every question we asked had the form "is this assumption discharged by the rest of the VM?" Some assumptions (the lookup table abstraction) seemed plausible, though unverified, while others (that memory interactions are syntactically time-ordered) were less clear. Instead of continuing to read and discuss, we decided to use verified code generation to reach consensus faster, by changing the specification towards something we all had more confidence in.

4 Methodology 1: Improving the specification

So, we changed our role as auditors: rather than argue about whether an assumption is acceptable, we changed it into one that was more acceptable to everyone. We did this for admissibility, whose order-sensitivity was just an artifact of how the memory discipline was phrased. An order-free phrasing also works: at every memory address, the multiset of received data exceeds the multiset of sent data by at most one element—the record that enters the chip from outside. This phrasing is invariant under permuting the interaction list.

Changing the codebase is rare in an audit, and traditionally infeasible for two reasons. First, an auditor lacks the expertise to change someone else's codebase. Second, the change could invalidate the audit's existing progress, forcing a restart from scratch. Both objections weaken under verified

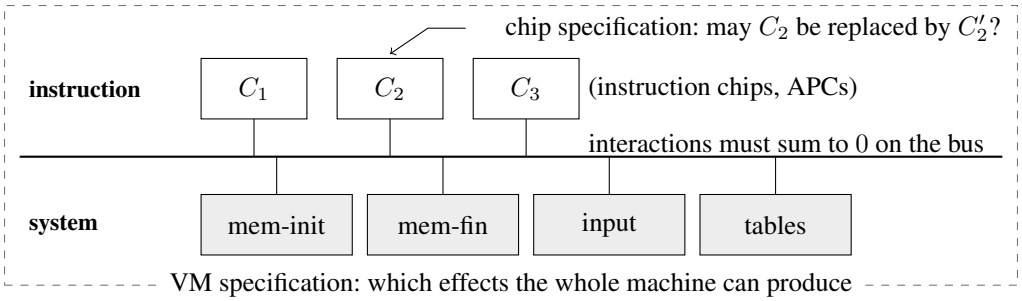

Figure 2: The two levels of specification. The chip specification relates one instruction chip to its replacement, abstracting everything outside it into per-message predicates. The meta-specification of §5 instead names the system chips and states correctness over the whole machine, which lets us prove that the chip specification implies it.

code generation. AI made the changes to the implementation, and we only took responsibility for the specification change; Lean then indicated exactly what proofs broke, AI fixed them, and we did not need to restart the audit. The division of labor was very effective: the auditors changed the specification, AI fixed the implementation and proofs, the developers noticed that this hurt performance, and then they guided AI to recover it, with no further specification change.

However, "improving" the specification has no stopping criterion. We made it more acceptable to us, but had no way to say when it was acceptable enough. We wanted an objective answer, and for that we needed something to check the specification *against*.

5 Methodology 2: Creating a meta-specification

To settle the specification's correctness, we widened its scope (Figure 2): we wrote a meta-specification, a specification for the whole VM. We specified each system chip, and proved that the existing chip-level specification implies the VM-level one. This was helpful, because we found the VM-level specification much simpler than the chip-level one.

The VM-level specification Writing the VM-level specification was easy. VM satisfiability is already clear: it is Eq. (1). All that remains is to define the VM's effects. Fix the system chips, and let $\mathbf{C}$ be a list of instruction chips. The VM's observable effect is the net bus contribution of the system chips that touch the outside world: the memory boundary chips and the input stream. Writing VmSat for Eq. (1) plus the instance limit, and $\mathsf{VmEffects}(\mathbf{a})$ for the net observable effects,

$$\mathsf{CanProduce}(\mathbf{C}, e) \triangleq \exists \mathbf{a}.\ \mathsf{VmSat}(\mathbf{C}, \mathbf{a}) \wedge \mathsf{VmEffects}(\mathbf{a}) = e,$$

$$\mathsf{Equivalent}(\mathbf{C}, \mathbf{C}') \triangleq \forall e.\ \mathsf{CanProduce}(\mathbf{C}, e) \iff \mathsf{CanProduce}(\mathbf{C}', e).$$

Note that VmSat only contains conditions checked by the cryptography. It intentionally excludes $\mathsf{Legal}(C)$ [26, `Legal.lean`], which makes the instruction-chip conventions of §2 precise; the VM cannot check it, because it quantifies over all assignments of C. It also excludes invariants that hold only as a consequence of several chips together, such as in-set lookup data and timestamp non-overflow. In our development, these became theorems rather than assumptions.

Specifying the system chips Specifying the system chips was harder, but still feasible. They are complex, and we do not own them. However, our approach only required us to specify them, not to verify them or to study their implementations in detail. That was essential, since every chip, system chips included, is implemented by algebraic terms that can be intricate, while its specification is usually simple. For example, a lookup table for the range $[0, 2^{29}) \subset \mathbb{F}$ algebraically breaks values into limbs to range-check them indirectly but efficiently, yet its specification is trivial: any received data d on the bus must satisfy $0 \leq d < 2^{29}$. Other system chips (like memory initialization) were more complex, but still quite manageable.

Proving the theorems The main theorems [27, `Theorems.lean`] say that if every chip in $\mathbf{C} \cup \mathbf{C}'$ is Legal, and each C_i' is a sound (resp. complete) replacement for C_i by the chip-level specification,

Table 1: What the reader has to read: lines of Lean, code and doc-comment counted separately, with `tokei`. The meta-specification's statement is slightly larger than the chip specification, but we found it easier to audit, and untrusted code far outweighs trusted.

Layer	Contents	Code	Doc	Audited?
VM specification	VM semantics, system model, Legal, theorems	492	606	yes
VM-level proofs	behind the VM-level theorems	5,472	1,371	no
Chip specification	chip equivalence, with assumptions	426	335	now derived
Optimizer	passes and their correctness proofs	39,776	5,509	no
Legality corpus	§6's analyses and APCs	5,224	1,014	statements only

then $\mathbf{C}'$ is a sound (resp. complete) VM-level replacement for $\mathbf{C}$. Given the statements and the system model, the AI did these proofs on its own over roughly 40 hours. The only human guidance was a LaTeX manuscript about why we thought the theorem was true, plus occasional corrections to our system model or Legal definition.

The payoff is that the chip-level specification becomes *untrusted*. It is absorbed into the implementation as a useful intermediate abstraction, which an AI could refactor later, so an auditor need only read the VM-level statement and the system model (Table 1). Our VM-level specification is actually longer than the chip-level one, but easier to audit, because almost every part of it corresponds to a concrete module that we could compare it against. (The one exception is Legal, which we address below.) The specification no longer makes assumptions that are justified only by global, inductive arguments.

Specifying both directions Generally, applications of zkVMs are more concerned with soundness than completeness. Accepting a false proof could be irrevocably exploited, while rejecting a good proof only stalls the system, which could be patched. However, under verified code generation, specifying only soundness is dangerous: soundness alone is trivially satisfied by a chip with no behaviors, and completeness alone by a chip with all behaviors. So, if we specified only one direction, AI could satisfy it degenerately. We stated both. As we hoped, each direction constrained the formalization differently, each exposed different defects in our specification, and ultimately we proved both.

Improving the chip-level specification Proving the theorems required two changes to the chip specification. The first was the order-free admissibility of §4. The second required adjacent instructions in an APC to have consecutive timestamps, which ensures that they execute in order. We do not know that the optimizer is incorrect without these changes; we needed them for AI to find a correctness proof easily. So, we view them as improvements to the assurance case.

Remaining assumptions The meta-specification left two assumptions standing. The first is the system model, which we did not check against OpenVM's real system chips. In our opinion, it is likely correct, since it is simple. Still, checking it is a verification problem, and we decided that it is not ours: the owner of a module should verify it against a specification, and we do not own OpenVM. Our contribution is to give its owners that specification. The second assumption is that each chip C given to powdr satisfies $\mathsf{Legal}(C)$. Since powdr's inputs are OpenVM instruction chips, the ownership argument above arguably says that this is not our responsibility either. However, the definition of Legal is intricate—not simple—so we wanted evidence that we had defined it correctly, through a form of *dynamic validation*.

6 Methodology 3: Dynamically validating the meta-specification

$\mathsf{Legal}(C)$ is a property of an individual chip, so it can be checked chip by chip; this is its most attractive feature as an assumption. So, we took a corpus of APCs from powdr's test suite, both before and after optimization, and imported them into Lean terms with a generator script.

Direct proofs of legality Our first attempt was to have AI prove each chip legal, but this was too slow. The chips are small by APC standards (10 to 30 bus interactions), but very large as Lean terms.

Table 2: The APC corpus, at the stage powdr's optimizer emits. Each is checked legal by one kernel `decide`, with a pairing of memory sends with receives and a per-interaction clock offset as hints. The corpus is small, but it refuted three clauses of Legal. Check time is the median of three runs on an Intel Core i7-8550U, minus import loading; it excludes elaborating the APC's circuit term, which is generated data, not proof.

APC	Bus interactions	Memory accesses	Clock ticks	Check time (s)
`SingleBeq`	10	2	2	1.4
`AndBranch`	15	3	5	1.5
`SingleXor`	18	3	3	2.6
`LoadBranch`	20	4	5	2.6
`Keccak2105000`	23	5	11	5.8
`TwoLoads`	30	6	8	8.8

AI could draft candidate proofs, but Lean could not check them, not even in ten minutes per chip. Even minute-long checks would be a problem, since they slow the AI's iteration loop.

Computation instead of proof Instead, we had AI write *decidable static analyses* for most of legality's conditions: Boolean functions over the interaction list, each with a proof that a `true` result implies the corresponding clause of Legal. Then, one `decide` per chip has Lean's kernel re-run the computation and rebuild the proof—*proof by reflection* [3]. Two properties of this arrangement made it work. First, the analyses had to be sound (`true` implies Legal), but not complete (`false` implies ¬Legal); they only had to be complete enough for real APCs, which are highly structured. When one did fail on some APC, it was easy to improve its completeness with AI. Second, some of the analyses take hints. For example, part of legality requires a pairing between memory sends and receives that satisfies certain conditions. Our analysis takes the pairing as an input and soundly checks the conditions. The pairing is typically easy to read off the chip informally, so AI provided it. Of course, the hint is untrusted by design: a bad hint hurts completeness, not soundness. A sound checker with an untrusted hint is the pattern of *certifying algorithms* [16]. Together, these two properties let us benefit from AI without having it generate huge proofs that Lean would struggle to check. Others have applied property-based testing [32, 4] and symbolic testing [12] to AI-generated specifications. What is new in our project, to our knowledge, is validating an AI-written specification by proving theorems about programs with an AI-written, Lean-checked analysis. Table 2 summarizes the corpus of APCs that we proved Legal.

Bugs found by validation This validation was very useful, because it revealed several specification bugs in Legal, each of which made our definition too narrow to apply to real APCs. First, timestamps were accidentally constrained in an infeasible way. Second, the specification required all interactions to have distinct times, but real APCs do have multiple interactions at the same time, on different buses; we relaxed the requirement, AI easily fixed the proofs, and the validation went through. Third, we required all memory sends to occur strictly between a chip's received and sent execution-bridge timestamps, but the interval is inclusive on the bottom end—another easy fix.

Without these fixes, our theorems would have held only vacuously on real APCs, since Legal is one of their hypotheses; this problem is classic in hardware verification, where it is called *antecedent failure* [1]. The result was a definition of Legal that we had more confidence in, because six real APCs satisfy it. Our confidence could be improved further: we could (1) validate with a larger corpus of APCs, (2) verify that all OpenVM instruction circuits are legal, (3) prove that a concatenation of legal chips is legal, and (4) prove that the optimizer maintains legality.

7 Related work

Verified code generation The term *autoformalization* usually refers to translating informal mathematical statements into formal ones, judged by fidelity [24]; several recent projects apply the same tools to code. Rinard [29] autoformalizes a compiler in Lean with Claude Code, Klaus et al. [9] report on a Rust-to-Lean pipeline for production cryptography, and Kan et al. [8] discharge thousands of lemmas without fixed proof strategies. All three take the specification as given, whereas our concern is the specification itself. Tafese et al. [31] autoformalize Stellar's SDEX offer-exchange subsystem,

written in C/C++, into PVS. In our case, the original code was instead ported to Lean by AI, to be correct by construction.

Auditing the specification Ho et al. [6] face our problem, auditing a specification that an agent helped draft, and solve it differently: they keep the Lean specification close to the published standard, make it executable, and run it on test vectors. Both moves require an external artifact to compare against, and we had none—there is no standard for a zkVM chip, and a chip specification is a relation on assignments, not a function to execute. Still, our approach resembles theirs: §6 is similar to evaluating test vectors. More broadly, Lahiri [13] argues that formalizing intent is the bottleneck for reliable AI coding, and proposes judging a generated specification by its own soundness and completeness. This is the same tension that we hit in §5, one level up.

Compilers and assumptions Verified compilers have always had to choose a correctness definition carefully [14], and for zero-knowledge compilers the choice is subtle enough to be a contribution on its own [22]. Prior work on this optimizer discharged correctness per-APC with an SMT solver [11]. The Lean port replaced that translation validation with a proof for all inputs, which made auditing the definition urgent. Our §6 returns to per-instance checking, but for the *specification's assumptions* rather than the compiler's output. This is a use of translation validation [23, 15] that we have not seen before.

Our meta-specification is a rely-guarantee decomposition [7]: Legal is the guarantee that each instruction chip owes the machine, and the system model is what it may rely on. Large verification efforts have long had to state such assumptions explicitly [10]. What has changed is the cost: formalizing the context used to be a large project in itself, and with AI it was cheap enough to do as part of an audit.

8 Discussion

Verified code generation moves the risk into the specification. For a module, that risk lies in what the specification assumes about its context; every difficulty in §3 was like this. Our experience suggests four further lessons.

Formalizing the context A global specification can be easier to write and audit than one for a single module. We found our VM-level specification considerably easier to audit than our chip-level one, so formalizing the context was the cheaper option.

Specifying both directions In our case, optimizer soundness and completeness are in tension. Each is trivially satisfiable on its own, so specifying both is essential. Otherwise, the specification is a degenerate target, and AI will satisfy it degenerately.

Specifications for neighboring modules We did not verify neighboring modules (e.g., OpenVM). Instead, we wrote specifications for them to verify against. We believe that this is the only approach that scales, and it also puts each assumption in front of the people who can discharge it.

Checkable assumptions We found it very helpful to make our assumptions *checkable*; this is how we learned that our first definition of Legal was vacuous. Validating assumptions by running code is an old practice: fuzzing [18] and cross-testing [17] check that a program's assumptions hold on the inputs it actually receives. These techniques work when an assumption quantifies over a program's inputs, because each run samples the quantifier. Our setting is one order up: the optimizer's inputs are *programs*, and Legal quantifies over *their inputs*, the assignments. So a single sample of the outer quantifier is still a universally quantified claim, and checking it is not an execution but a proof. Each test case is a theorem. What makes dynamic validation practical again is that AI can discharge those theorems—either directly, or, when the proofs are too large to check (§6), by writing sound decidable analyses that reduce each theorem to a computation.

Acknowledgments and Disclosure of Funding

We thank Alex Hicks and the Ethereum Foundation for funding this project.

References

[1] Derek L. Beatty and Randal E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *DAC*, 1994.

[2] Manuel Blum, William Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. *Algorithmica*, 12:225–244, 1994.

[3] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *TACS*, 1997.

[4] Yueyang Feng, Dipesh Kafle, Vladimir Gladshtein, Vitaly Kurin, George Pîrlea, Qiyuan Zhao, Peter Müller, and Ilya Sergey. Certified program synthesis with a multi-modal verifier. arXiv:2604.16584, 2026.

[5] Frommi. `miniz_oxide`: a pure-Rust replacement for the miniz deflate/zlib encoder and decoder. `https://github.com/Frommi/miniz_oxide`. Rust crate.

[6] Son Ho, Cédric Fournet, Antoine Delignat-Lavaud, Samuel Lee, Jason Fisher, and Jessica Krynitsky. Verifying Rust cryptography in SymCrypt, from standards to code. `https://www.microsoft.com/en-us/research/blog/verifying-rust-cryptography-in-symcrypt-from-standards-to-code/`, July 2026. Microsoft Research blog.

[7] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[8] Shuangxiang Kan, Shuanglong Kan, and Sebastian Ertel. Harnessing code agents for automatic software verification. arXiv:2607.06341, 2026.

[9] Natalia Klaus, Juan Conejero, and Palina Tolmach. A Rust-to-Lean verification pipeline with AI provers: An experience report. arXiv:2605.30106, 2026.

[10] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *SOSP*, 2009.

[11] Gereon Kremer, Alex Ozdemir, and Arie Gurfinkel. Formally verifying powdr's autoprecompiles. `https://www.certora.com/blog/formal-verification-powdr-autoprecompiles`, August 2026. Certora blog.

[12] Shuvendu K. Lahiri. Evaluating LLM-driven user-intent formalization for verification-aware languages. In *FMCAD*, 2024.

[13] Shuvendu K. Lahiri. Intent formalization: A grand challenge for reliable coding in the age of AI agents. `https://risemsr.github.io/blog/2026-03-05-shuvendu-intent-formalization/`, March 2026. Microsoft Research RiSE blog.

[14] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[15] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for LLVM. In *PLDI*, 2021.

[16] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.

[17] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[18] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[19] Kim Morrison. Why Lean is faster than Rust. `https://kim-em.github.io/blog/2026-7-24-why-lean-is-faster-than-rust/`, July 2026. Blog post.

[20] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *CADE*, 2021.

[21] OpenVM Contributors. OpenVM: a modular zkvm framework. `https://github.com/openvm-org/openvm`, 2025.

[22] Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Clark Barrett. Bounded verification for finite-field-blasting in a compiler for zero knowledge proofs. In *CAV*, 2023.

[23] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *TACAS*, 1998.

[24] Auguste Poiroux, Gail Weiss, Viktor Kunčak, and Antoine Bosselut. Reliable evaluation and benchmarks for statement autoformalization. In *EMNLP*, 2025.

[25] powdr Labs. A verified optimizer for powdr autoprecompiles, in Lean. `https://github.com/powdr-labs/apc-optimizer`, 2026. Commit `1b3eb07`.

[26] powdr Labs. Legality of guest chips for the powdr optimizer. `https://github.com/powdr-labs/apc-optimizer/blob/1b3eb07d7aeda433c18fc1fec40f8d21e01fff5d/ApcOptimizer/VmSpec/Legal.lean`, 2026.

[27] powdr Labs. VM-level correctness theorems for the powdr optimizer. `https://github.com/powdr-labs/apc-optimizer/blob/1b3eb07d7aeda433c18fc1fec40f8d21e01fff5d/ApcOptimizer/VmSpec/Theorems.lean`, 2026.

[28] powdr Labs. powdr. `https://github.com/powdr-labs/powdr`, 2026.

[29] Martin Rinard. Testing, credible compilation, and verification in the axon verified compiler in lean and claude code. *arXiv preprint arXiv:2605.01660*, 2026.

[30] Succinct Labs. SP1: a performant, open-source zkvm. `https://github.com/succinctlabs/sp1`, 2025.

[31] Joseph Tafese, Karthik Nukala, Hassen Saïdi, Natarajan Shankar, Arie Gurfinkel, and Giuliano Losa. Show me the money: An exercise in proof-driven software understanding. In Eva Darulova, Anthony W. Lin, and Philipp Rümmer, editors, *Computer Aided Verification - 38th International Conference, CAV 2026, Lisbon, Portugal, July 26-29, 2026, Proceedings, Part III*, volume 16684 of *Lecture Notes in Computer Science*, pages 230–243. Springer, 2026. doi: 10.1007/978-3-032-32537-2_11. URL `https://doi.org/10.1007/978-3-032-32537-2_11`.

[32] Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. VERINA: Benchmarking verifiable code generation. arXiv:2505.23135, 2025.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: In this paper, we simply report our experience auditing a specification via formalization and reflect on which methodologies worked and why. That is what the abstract says.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We are careful to only make claims about our own experience—clearly our experience might not generalize.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [Yes]

Justification: Our main results are theorems, machine-checked in Lean (§5). Their full statements and proofs are in our public repository [25], including the main theorems [27] and the definition of Legal [26].

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: The project is open-source [25]; our results are from commit `1b3eb07`. The only experimental results are the check-times of certain proofs in that tree.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: See above.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [N/A]

   Justification: we train no models.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [N/A]

   Justification: we conduct no hypothesis tests

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).

- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: we state our hardware for the experiment we report.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: our work involves no human subjects; the knowledge it might create is likely to be only be applicable to making it easier to create verified software.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: we feel that our work will have little direct impact on societal dynamics.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.

- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: we have no datasets or models.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: we use three assets: Claude, Lean, and powdr, which are all acknowledged.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.

- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: no new assets.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: these are not involved

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: these are not involved.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: our work uses LLMs extensively, as we describe.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.