Agentic Model Checking

Youcheng Sun*, Jiawen Liu*, Humaid Alqasimi*,
Lucas Cordeiro†, Daniel Kroening‡, Jason Xue*
*Mohamed bin Zayed University of Artificial Intelligence (MBZUAI)
†University of Manchester ‡Amazon
techieyoucheng@gmail.com, Jiawen.Liu@mbzuai.ac.ae, Humaid.Alqasimi@mbzuai.ac.ae
lucas.cordeiro@manchester.ac.uk, daniel.kroening@gmail.com, Jason.Xue@mbzuai.ac.ae

***Abstract*—Verifying large software systems is difficult. Formal specifications are rarely available, and the assumptions code relies on are often encoded implicitly at call sites rather than enforced at function boundaries. This paper presents *agentic model checking*, a verification scheme that combines LLM agents with bounded model checking (BMC) under the principle that *agents propose, solvers verify*: agents perform semantic reasoning such as specification inference and verification planning, while every soundness-critical decision remains with the formal backend. We realize this scheme in BMC-Agent, an agentic verification system built on an off-the-shelf bounded model checker.**

The key contribution of this paper is *agentic orchestration* of the model checker. Rather than fixing a single verification recipe, BMC-Agent plans how bounded model checking should be applied to each verification task. It infers function specifications, configures the verification task, decomposes the analysis compositionally using specification-constrained stubs, scopes whole-program verification to property-relevant code, validates counterexamples through deterministic checks and replay, and iteratively refines specifications in response to spurious witnesses. This orchestration enables the same BMC engine to scale from individual functions to large software systems while preserving solver-backed verification.

We evaluate BMC-Agent on LLM-generated operating-system and compiler code, standard verification benchmarks, and mature OSS-Fuzz-hardened libraries. Across these workloads, BMC-Agent discovers confirmed defects in heavily tested software and produces bounded clean verifications without requiring manually written specifications. These results demonstrate that agentic orchestration substantially extends the practical scalability of bounded model checking for real-world software.

***Index Terms*—bounded model checking, LLM agents, specification inference, compositional verification, systems software**

I. Introduction

LLM coding assistants now generate substantial bodies of systems code [1], operating systems, compilers, drivers, and embedded utilities, in both C and increasingly Rust. Studies report a high number of incidents of security-relevant defects [2], [3]. Verifying the code is hard: the developers who deploy it rarely understand every function, manual specification is infeasible at scale, and the bugs that arise are predominantly low-level, integer overflows, off-by-one indices, and missing bounds checks on malformed input, which require exhaustive reasoning over all inputs to rule out. An LLM can often flag a suspicious line; it cannot prove the bug absent.

A second pattern recurs across codebases: LLMs encode safety contracts *implicitly* at call sites rather than enforcing them at function boundaries. Public byte-helpers (slice readers, offset-arithmetic writers) omit bounds and overflow guards, while every in-tree caller happens to maintain the missing invariant. The code works on well-formed input and fails on crafted input through the same public functions, even though no current call site triggers it. A verifier that reports errors without distinguishing in-tree reachability from public-API ingress either over-reports or under-reports; useful verification needs explicit framing of which threat model is in scope and which bugs are active versus latent.

The verification community has techniques for exactly these bugs. Bounded model checking (BMC) [4] checks whether a property is violated within an execution of at most k steps; CBMC [5] applies it to C, Kani [6] to Rust, and JBMC [7] to Java bytecode, each returning a concrete counterexample or a bounded proof. Compositional verification [8] and CEGAR [9] scale and refine such checks. But each assumes what LLM-generated code lacks: a specification, a property catalog, or author-written annotations. Recent work addresses this gap using LLMs themselves [10]–[13]. LLM-only approaches scale well and provide useful explanations, but are unsound for arithmetic and memory-safety properties that require exhaustive path reasoning. LLM-assisted deductive verification provides stronger guarantees, but typically produces no concrete counterexamples when verification fails.

Agentic model checking: We couple LLM agents with a *bounded model checking* backend under the principle *agents propose, solvers verify*. Agents do the semantic work, inferring specifications, selecting which checks each function warrants, classifying counterexamples, and proposing refinements, while every output that could change a verdict passes through a deterministic check: a specification parser, a BMC query, a solver-based soundness guard, or concrete execution. BMC suits this arrangement for three reasons: it yields concrete counterexamples that a classifier can replay against the real program rather than guess a verdict; it checks each function in isolation against its specification, with callees replaced by postcondition stubs, so verification parallelizes across the call graph; and a precondition tightened to eliminate a spurious witness immediately sharpens every caller, since that specification is reused as the callee's stub, CEGAR-lifted to the specification level and propagated compositionally.

This paper makes the following contributions:

1) We present *agentic model checking*, a scheme in which LLM agents *orchestrate* a bounded model checker, decid-

ing how the checker is applied to each program, while every soundness-critical decision remains solver-backed;
2) We design and implement BMC-Agent, a tool that orchestrates an existing bounded model checker by inferring specifications, selecting and configuring checks, deciding what to inline versus stub, scoping the harness, validating counterexamples, and refining iteratively, so the same engine becomes both sound on unspecified code and scalable to whole systems. Compositional contract inference is one of these orchestrated decisions, not the whole method;
3) We evaluate BMC-Agent on diverse codebases, demonstrating confirmed bug finding and bounded clean verification on large-scale and heavily tested software.

II. Motivating Example

The hard part of verifying real code is not noticing a suspicious line; it is recovering the contract the author never wrote down, telling safe-but-unguarded code apart from a real defect, and doing so across the call chain.

Consider a small packet parser (Listing 1), representative of a common pattern in real code, where the key issue is not inside any individual function but at the public boundary. This is a minimal example we construct to make the mechanism visible in a few lines, not one of the findings reported later; the unenforced-contract pattern it isolates is what those findings exhibit at scale.

a) Recovering the contract by refinement: `read_u32` has no bounds check, and it looks dangerous when read in isolation: an LLM asked to audit it will flag the unguarded reads, or "fix" them with a length check every caller would then have to supply. Both are wrong. `read_u32` is correct under a precondition its callers are meant to honour, that `p` addresses four readable bytes, which nothing in the source states. BMC-Agent does not guess that contract in one shot; it converges to it by letting the solver correct the agent:

1. *propose*: the spec agent offers the only precondition the call sites alone justify, `p != NULL`;
2. *check*: under it, BMC analyses `read_u32` in isolation and returns a counterexample in which a non-null `p` addresses fewer than four readable bytes, causing an out-of-bounds read at `p[3]`;
3. *validate*: the witness is genuine but unreachable from any observed in-tree call site, so it is not an active defect. It enters the refinement loop (§III-D) as a contract candidate rather than being reported as a crash;
4. *refine*: the refiner tightens the precondition to `valid_range(p, 0, 4)`. The soundness guard accepts it only after checking that every observed call site implies the strengthened precondition, so the refinement excludes no reachable state;
5. *re-check*: BMC proves `read_u32` safe under the refined contract, which then propagates to its callers as a stub.

Neither the LLM nor the solver reaches the answer alone: the LLM proposes the contract the source omits but cannot certify it, and the solver certifies contracts but has none to check until the agent supplies one. The solver alone decides whether a contract holds, and it keeps sharpening the agent's proposal until the function either verifies or yields a witness that no sound refinement can remove.

Listing 1: A length-prefixed packet parser. `read_u32()` is an exported helper that assumes four readable bytes. All in-tree callers satisfy that assumption, but it is not enforced at the public boundary.

```
1
2  uint32_t read_u32(const uint8_t *p) {
3      return p[0] | (p[1] << 8) | (p[2] << 16) | (p
           [3] << 24);
4  }
5
6  static uint32_t decode_header(const uint8_t *buf,
       size_t len) {
7      if (len < 8)
8          return 0;
9
10     uint32_t count = read_u32(buf + 4);
11
12     if ((uint64_t)8 + (uint64_t)count * 4 > len)
13         return 0;
14
15     uint32_t sum = 0;
16     for (uint32_t i = 0; i < count; i++)
17         sum += read_u32(buf + 8 + i * 4);
18
19     return sum;
20 }
21
22 uint32_t parse_packet(const uint8_t *buf, size_t
       len) {
23     if (len < 8 || read_u32(buf) != PKT_MAGIC)
24         return 0;
25
26     return decode_header(buf, len);
27 }
```

b) Compositional verification, and why nothing here crashes: The contract settles only one of two questions: whether the program crashes as it uses `read_u32` today. With the contract recovered, BMC-Agent checks each caller against it, replacing `read_u32` with its postcondition and asserting `valid_range(p, 0, 4)` at every call. `decode_header` bounds `count` against `len` before the loop, so every `read_u32(buf + 8 + i*4)` stays in range; `parse_packet` forwards a length it has already checked. The solver proves the obligation at every in-tree call site: no execution reachable through the internal call graph violates the contract. There is no active bug, and an LLM that flagged `read_u32` would have reported a false positive.

c) Latent at the boundary: The second question is the public boundary, and the in-tree proof does not answer it. Two functions are exported, `read_u32` and `parse_packet` (`decode_header` is file-local), and they behave differently at that boundary. `parse_packet` takes the buffer length and checks it before any read, so it is safe against an arbitrary external caller, and BMC-Agent verifies this property. `read_u32` takes only a pointer: its signature carries no length, so it cannot check anything, and its safety rests entirely on an obligation its in-tree callers happen to meet. An

external caller handing it fewer than four readable bytes causes an out-of-bounds read, a state no in-tree caller produces. Under the security threat model BMC-Agent therefore reports `read_u32`, and not `parse_packet`, as a *latent* exposure, distinct from an active crash: safe as the program uses it today, but reachable through the public API by a caller the source does not contain.

d) Why this needs BMC-Agent: An LLM can name the exposure but cannot prove that the in-tree callers all satisfy the contract; a model checker can prove that, but without the inferred contract and the in-tree-versus-public distinction, it flags every unguarded read or none. Recovering the unwritten contract, proving that every caller honours it, and isolating the one place it is exposed but unenforced is exactly what BMC-Agent is built to do.

III. BMC-Agent Architecture

Figure 1 presents the BMC-Agent architecture. The system combines LLM-driven semantic inference with a deterministic bounded model checker under the principle that *agents propose, solvers verify*. Its core idea is *agentic orchestration of the checker*: the BMC engine is fixed and off-the-shelf, and an agent decides, per program, *how* to apply it: which specifications to infer, which checks and solver configuration each function warrants, what to inline versus replace with a constrained stub, how to scope the harness, and how to validate the resulting counterexamples. Driven this way, one engine reaches different operating points: precise on small, unspecified functions, and tractable on large whole-program inputs. Execution begins by parsing the source code, building the call graph, and constructing a compact codebase-wide domain summary from headers, types, function signatures, and optional external domain material such as API documentation or hardware specifications.

Orchestration spans the full analysis: specification inference, per-function check and solver configuration, inline-versus-stub and harness-scoping choices, counterexample validation, and refinement. One such decision is *compositional decomposition*: rather than posing a whole-program verification query, the agent verifies each function against an inferred contract, replacing callees with stubs constrained by their postconditions, which bounds each BMC query by a single function rather than the entire codebase and lets verification proceed in parallel across call-graph layers. When a whole-program entry must instead be checked directly, the agent scopes the query (inlining the call paths relevant to the property and constraining the remainder) so the formula stays within the checker's resource limits. Specification inference is thus one of several orchestrated decisions.

Agents propose, solvers verify: BMC-Agent delegates open-ended semantic judgments to LLM *agents* and reserves verification judgments for deterministic components. An agent is an LLM-driven workflow that may retrieve program artifacts, invoke analysis tools, inspect intermediate verification results, and produce a structured proposal such as a specification, a check or configuration selection, a scoping or inline-versus-stub decision, a realism assessment, or a refinement suggestion. Agents never produce verification verdicts directly. Everything that must be reproducible is deterministic: the BMC solver itself, translation of the specification notation into solver assumptions and assertions, counterexample deduplication, reachability propagation, and the refinement soundness guard.

Table I summarizes the agents used by BMC-Agent and the deterministic mechanisms that check or constrain their proposals. In BMC-Agent, agents contribute candidate specifications, analyses, and refinements; deterministic components decide whether those proposals are accepted, rejected, or revised. Every reported bug must ultimately be supported by the evidence generated during verification and validation.

A. Specification Inference and BMC Configuration

One orchestration stage prepares each function for the checker, via two agents. A *specification* agent infers *what* the function must guarantee: its contract (§III-A1). A *BMC configuration* agent then decides *how* to pose the query (§III-A2): which checks to enable, the unwind and solver settings, and, reading the callees' real bodies, which to keep exact (inline) versus abstract (stub), the choice that determines the scope of exact reasoning in each query. This stage outputs a per-function contract plus a verification configuration that sets the checks, the solver options, and the inlining and scope.

1) Specification Generation: The specification generator traverses the call graph top-down, beginning from functions with no callers and proceeding layer by layer toward leaf functions. Generating specifications top-down derives a function's intended behaviour from how it is used by its callers rather than merely restating implementation details. For each function, the agent performs the following steps: (i) retrieves the function body, caller context, and domain summary; (ii) drafts a pre/postcondition pair in the assertion notation; (iii) validates the draft using a deterministic parser; and (iv) retries on failure up to a configurable limit.

Multi-perspective inference. BMC-Agent infers a function's contract by triangulating it from independent angles rather than trusting any single one, on the principle that the implementation is never its own oracle. Two angles target the *defensive* contract, meaning what must hold for the function not to fault, derived respectively from its *callers* (what the call sites assume and rely on) and from its *body* (what the implementation actually requires). Each has a blind spot: callers can mask a latent requirement by only ever passing safe values, and the body can encode a bug as if it were intended behaviour. BMC-Agent therefore drafts the defensive specification twice, a first LLM call emphasizing caller intent and a second emphasizing implementation behaviour, then verifies against the stronger of the two and flags functions where the drafts *disagree*, since a caller-versus-body conflict points at a latent requirement, a misuse, or a genuine bug. The specification of a flagged function is re-derived with additional caller context retrieved on demand, so the contradiction is resolved with more evidence.

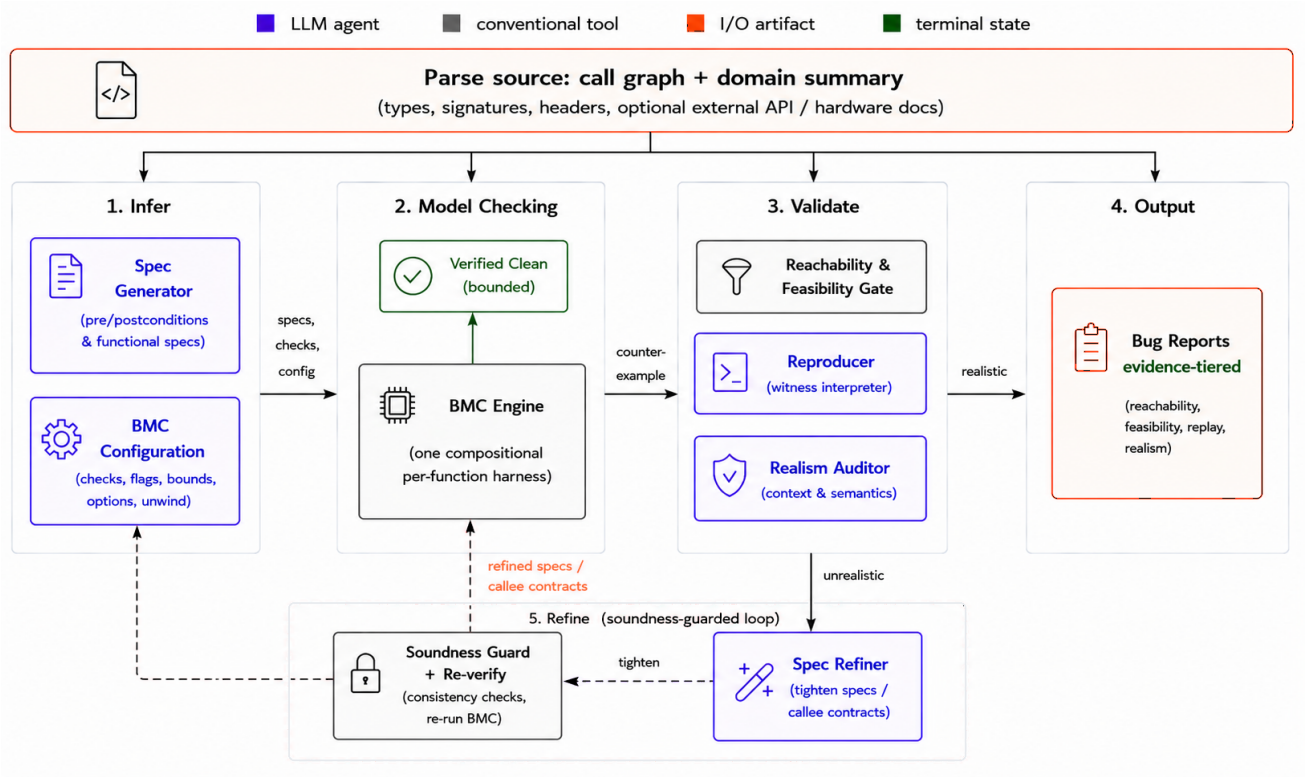


Fig. 1: BMC-Agent architecture.

TABLE I: The LLM agents in BMC-Agent. No agent emits a verdict: the BMC engine (§III-B1), deliberately *not* an agent, decides whether a counterexample exists, and the validation pipeline gates every reported bug.

Agent	Stage	Proposal	Deterministic safeguard
Spec Generator (§III-A1)	infer	Function body, caller context, domain summary → pre/postconditions and optional functional specifications	Parser rejects malformed specs; the solver is the arbiter of a spec's consequences; over-tight specs are caught in refinement
BMC Configuration (§III-A2)	infer	Function and callee bodies, call graph → checks and solver settings, and per-callee inline-versus-abstract (scope)	Conservative default on failure (affects coverage, not soundness); property-reaching callees always inlined (a sound floor), so over-abstraction only adds spurious witnesses, caught in validation and refinement
Reproducer (§III-C)	validate	Witness, call chain, source context → a system-entry harness driving the witness inputs from the highest available caller down to the fault site (entry function, includes, linkage)	The harness must compile and actually fault under GCC/ASan; the captured signal is the verdict, so the agent cannot promote a non-faulting witness, only fail to reproduce a real one
Realism Auditor (§III-C)	validate	Specification, harness, witness state, source context → realistic or unrealistic assessment	Deterministic witness-pattern pre-filter; the verdict is recorded and re-checked in refinement, not final on its own
Spec Refiner (§III-D)	refine	Witness, harness, source → tightened preconditions or callee contracts	Soundness guard and re-verification

A third, optional angle targets *behavioural faithfulness*: whether the function computes what it is meant to, not merely whether it avoids faulting. From the body, name, and signature, the generator infers the intended result and states it as a reference, typically a reference-equivalence expression (`result == buf[0] | buf[1]<<8 | buf[2]<<16 | buf[3]<<24` for a little-endian reader), an algebraic identity (`result % a == 0`), a fold over a sequence (a checksum equals the sum of its bytes), or a round-trip property (`decode(encode(x)) == x`); any boolean the backend can verify within the bounded model is admissible. It is conjoined with the postcondition, so the same BMC query checks it. Because the reference is inferred and bounded-checked, a pass means *bounded* faithfulness, not proof; the value is in disagreement. A divergence between reference and body is flagged as a discrepancy (a wrong-value bug, or a wrong reference) and adjudicated downstream.

Specification notation. Specifications are written as pre- and postconditions in a simple assertion notation: a few memory-safety predicates (`valid(p)`, `in_bounds(a,i)`, `valid_range(p,lo,hi)`) together with ordinary boolean and arithmetic expressions over the inputs and the result. The notation is deliberately minimal so that each clause translates deterministically into the backend's assume/assert primitives: a precondition becomes an assumption constraining the function's nondeterministic inputs, a postcondition becomes an assertion the backend must prove, and predicates expand to expressions in the program (`valid(p)` becomes `p != NULL`).

2) BMC Configuration: Checks, Inlining, and Scope: A single tool-using agent decides how the checker is invoked for each function, reading the function and, on demand, its callees' bodies. It settles three coupled choices. *Checks:* which optional backend checks to enable (e.g. unsigned- and signed-integer overflow, lossy type conversion, out-of-bounds pointer arithmetic) together with the unwinding and solver settings. *Inlining:* for each callee, whether to keep it *exact* (inline its body) or *abstract* it with a postcondition-constrained stub, or, lacking a useful contract, a nondeterministic havoc of its writable state. *Scope:* as a consequence of the inlining choice, how much of the call graph each query reasons about precisely versus bounds away. Choosing per function, and from the real bodies rather than a syntactic heuristic, avoids the overhead and noise of enabling every check and inlining the whole program, and reflects what each callee actually does for the property at hand.

The inlining choice is *promote-only over a sound floor*. A structural analysis marks every callee on a path that can reach the property and *always* inlines it; the agent may only *promote* further callees to exact, never abstract a floor function. Abstraction is thus confined to functions off the property-reaching cone, so it can add spurious behaviours (caught downstream by validation and refinement) but never remove a real violation: soundness does not rest on the agent's judgement. The agent runs in parallel across functions and falls back to a conservative minimal configuration on any failure, so verification is never blocked. This single configuration decision is also what allows one engine to scale: aggressive inlining for precision on small, unspecified functions, and tight scoping (inline the property-relevant cone, abstract the rest) to keep a whole-program query within the checker's reach.

B. Per-Function Model Checking with Compositional Stubs

Given an inferred specification and the per-function BMC configuration (checks, solver settings, and the inlining/scope choice), BMC-Agent runs one bounded model-checking query per function, inlining the callees the configuration kept exact and replacing the rest with postcondition-constrained stubs, then deduplicates the resulting counterexamples before reachability validation.

1) Verification Harness Construction: BMC-Agent builds one verification harness per function. Assembling the harness is a deterministic step. The harness initializes nondeterministic inputs, injects environmental assumptions, replays learned invariants from the verification knowledge base, translates the specification into backend assumptions and assertions, invokes the target function, and checks its postconditions alongside the selected safety properties.

Each query runs under a configurable unwinding bound k and per-function timeout. Bounded retries absorb the transient inconclusive results a backend can return: when a run exhausts a resource limit (encoding capacity, unwinding depth, or wall-clock time), BMC-Agent re-runs it with adjusted parameters (a larger encoding budget, a deeper bound, or, for state-heavy parser loops, a shallower one) before declaring the verdict inconclusive. The retries are finite, so the worst case yields an *inconclusive* verdict rather than blocking the pipeline.

Each query has one of three outcomes: (i) *verified clean*, (ii) *counterexample*, or (iii) *inconclusive*. A verified-clean result is recorded as a bounded, spec-relative verification artifact, and a counterexample proceeds to deduplication and validation. The backend exposes no *inconclusive* verdict of its own (CBMC reports only success or a property violation); BMC-Agent assigns it whenever a query neither proves the property nor returns a witness. This covers a timeout or encoding-capacity blowup that outlasts the retries, a backend front-end error, and a vacuous "success" that checked zero conditions (e.g. an unlinked function body), which a vacuity check demotes rather than counts as clean. Such a query is reported as inconclusive, never folded into verified clean, so an undecided or un-analysed function can never masquerade as proven.

Functions are dispatched in call-graph order, callees before callers, with parallelism within each layer, so that any contract resulting from a callee's verification is already in place when its callers are checked.

2) Counterexample Deduplication: A single defect frequently surfaces as many related counterexamples: the same root cause reached along different unwound paths. Before validation, a deterministic deduplication step merges these redundant witnesses into representative classes keyed by `(function, property type)`, so each defect is validated and reported once rather than once per path.

C. Counterexample Validation

A BMC counterexample is not immediately reported as a bug. For realistic code in a large-scale system this validation is the most consequential part of the workflow: whether a function is genuinely buggy, and whether a counterexample is real, depends on how the function is reached and constrained in context, not on the function in isolation. A witness that violates a property in a standalone harness may be unreachable, or require inputs no caller can supply; conversely, a function that looks defensible on its own may be the live ingress for a crafted input. After deduplication, each surviving witness therefore passes through four validation stages before emerging with an evidence tier.

Stage 1 - Callee feasibility: A BMC witness is computed with callees replaced by stubs, so it may depend on a stub return value no real implementation produces. Stage 1 substitutes

the real callee bodies, pins the witness inputs, and re-runs the backend; a witness that survives is a genuine execution of the function rather than an artifact of an over-permissive stub.

Stage 2 - Input reachability: BMC-Agent then propagates the surviving witness's inputs upward through the call graph, asking at each step whether the caller can supply the values the backend chose. The witness's reachability strength and evidence tier (CONFIRMED_BMC vs. CONFIRMED_SYSTEM_ENTRY) depend on where propagation halts: at a no-caller system boundary or mid-chain.

Stage 3 - Dynamic replay: For surviving witnesses, the *Reproducer* agent (Table I) builds a harness that drives the witness inputs to the fault site and executes it; a captured runtime fault upgrades the finding to CONFIRMED_DYNAMIC. Building the harness is an LLM proposal, but the verdict is not: a fault observed in real compiled code cannot be manufactured. Reproduction is sufficient evidence of a defect, not necessary: because the harness models only the witness state, many real bugs surface only on the deployed platform.

Stage 4 - Realism audit: Stages 1–3 establish that the witness is reachable in the model (symbolically, in Stages 1–2) and reproducible on a host harness (concretely, in Stage 3). Neither rules out witnesses that cannot arise in a running deployment for reasons outside the BMC model and the host harness, such as uninitialized framework globals, impossible hardware states, or path-divergent unwind artifacts. A final realism audit reads the specification, harness, witness state, and surrounding source, and classifies the witness as **realistic** or **unrealistic**.

D. Closing the Loop: Refinement and Persistence

A counterexample the validation stage cannot confirm is not discarded: it drives a model-sharpening loop whose accepted results persist for the remainder of the verification run.

Refinement and Re-verification Every witness that validation could not confirm as a real bug enters the refinement loop as *spurious*: not only those judged unrealistic in Stage 4, but also those unreachable from any caller (Stage 1) or resting on a callee stub return no real implementation produces (Stage 2). The loop does not suppress findings. It sharpens the verification model so the next BMC pass either confirms the bug, reaches a verified-clean result, or exposes a new witness class.

Refinement proposal. Given the witness, harness, and source context, an LLM proposes one of three types of remediation:

- *Verifier-level* (verifier artifacts, e.g. loop-unwind): adjust the BMC backend's parameters such as the unwinding bound, slice bound (Rust path), or active check selection.
- *Caller-side* (Stage 1 / Stage 4: inputs no caller supplies): tighten the function's precondition using properties enforced at every observed call site.
- *Callee/spec* (Stage 2: an impossible stub return): strengthen a stubbed callee's contract from its actual implementation or from sibling-function behaviour.

Soundness guard. Tightening a precondition is the dangerous move: a refinement that excludes too much would assume away a state a caller can actually reach, turning a genuine defect into a spurious verified-clean. The guard exists to stop exactly that. A proposed refinement is accepted only if it clears two checks: (i) the states it newly excludes must be unreachable from every observed call site, which BMC-Agent decides with the BMC backend where it can and otherwise with an LLM grounded in the actual callers; and (ii) it must not erase a counterexample already judged realistic, so any tightening under which a confirmed witness disappears is rejected as masking. A refinement that fails either check is discarded and its witness stays open. The guard never invents a verdict; it only decides whether the agent's proposed assumption is safe to adopt.

Re-verification. Accepted refinements trigger re-execution of the BMC backend under the new assumptions. Each function carries a bounded iteration budget; unresolved witnesses at the budget boundary are recorded for later review.

Persistence. Refinements and their verdicts persist to an on-disk store, so the verification is incremental: converged specs and learned callee contracts are reused on re-verification and resume rather than regenerated, and a curated library of witness-pattern detectors short-circuits the realism audit (§III-C) on recurring artifact classes before any LLM call.

E. Scaling BMC through Orchestration

The orchestration above is what lets BMC-Agent drive an off-the-shelf bounded model checker at the scale of a large, real-world codebase. Five mechanisms, each instantiated in the subsections above, keep every BMC invocation small while still covering the whole codebase.

Per-function decomposition under assume-guarantee. Each function is checked in isolation against its spec, callees replaced by stubs constrained by their postconditions (§III-B1): assume-guarantee at function granularity. No function's verification depends on another's body, so the encoded state space is bounded by one function's footprint, not the program's.

Call-graph-ordered parallel dispatch. Functions are verified callees-before-callers, so each callee stub is ready when its caller is checked; within a layer they are independent and dispatched concurrently (§III-B1), with check selection (§III-A2) running as a parallel pre-pass.

Witness deduplication before validation. A single defect often yields many related counterexamples; BMC-Agent merges them before validation (§III-B2), so the costly stages (callee inlining, dynamic replay, realism audit) run once per defect, not once per witness.

Agentic BMC configuration. Rather than treating bounded model checking as a fixed procedure, BMC-Agent performs *verification planning*: for each function, it decides how the verification query should be posed so that the property can be checked both soundly and efficiently. The BMC configuration agent (§III-A2) plans the verification strategy by selecting the appropriate safety checks, unwind bound, solver settings, and analysis scope, including when to inline callees for precise reasoning and when to replace them with abstractions. This

avoids the overhead of applying the same verification strategy uniformly across all functions.

Incremental persistence. A specification store persists tightened specs and reloads them on re-verification and resume, so refinement and repeated verification runs avoid regenerating specifications for functions already analysed; a pattern library short-circuits the realism audit (§III-C) on recurring artifacts before any LLM call.

These mechanisms decouple BMC's cost from program size: the per-function query scales with one function's state space, and the per-codebase cost scales with the call graph rather than the program text.

IV. Evaluation

We evaluate whether BMC-Agent's agentic orchestration lets sound bounded model checking scale to, and surface real defects in, code that monolithic verifiers cannot encode.[1] Our subjects span LLM-generated systems, mature open-source software, and standard verification benchmarks, and our study addresses four research questions:

RQ1: Scaling and comparison with traditional verifiers. Can agentic compositional decomposition let sound bounded model checking decide large, multi-function code that monolithic BMC cannot encode, and how does BMC-Agent compare to established automatic verifiers such as CBMC [5], ESBMC [14], and CPAchecker [15]?

RQ2: Comparison with agentic verification. Does BMC-Agent discover more confirmed defects than existing agentic verification systems on realistic codebases?

RQ3: Specification quality. Are the specifications generated by BMC-Agent competitive, as standalone artifacts, with specialized synthesizers?

RQ4: Real-world applicability. Can BMC-Agent produce actionable findings on mature open-source software and heavily fuzzed codebases?

A. Experimental Setup

BMC-Agent uses Claude Sonnet 4.5/4.6 as its agent backbone, CBMC 5.95.1 as the C verification backend, and Kani as the Rust verification backend.

We evaluate on four corpora spanning both generated and human-written software:

- **VibeOS** [16]: a 15 KLoC ARM64 operating system written with substantial LLM assistance.
- **Claude's C Compiler (CCC)** [17]: an approximately 50 KLoC Rust C compiler generated primarily through LLM-assisted development.
- **OSS-Fuzz and mature open-source software** [18]: including jq, OpenSSL, libcurl, libxml2, protobuf-upb, and additional manually selected targets.
- **SV-COMP benchmarks** [19]: a subset covering memory safety, arithmetic overflow, and reachability properties.

[1] All experimental data, configurations, and detailed results are available in the anonymous supplementary materials: https://anonymous.4open.science/r/artifact-r6v2k9m/.

B. RQ1: Comparison with Traditional Verification Tools

BMC-Agent's central claim is that agentic *orchestration* lets sound bounded model checking scale to multi-function code that monolithic model checkers cannot encode in a single query. We test this claim against established automatic verifiers on SV-COMP, which supplies ground-truth verdicts (each task's `.yml`) and a field of mature baselines.

a) Benchmarks: We evaluate on two tiers, both for the `unreach-call` property (reachability of `reach_error`), using the 64-bit (LP64) data model. **Tier A (BMC-tractable control):** a 27-task balanced sample of AWS-C-COMMON (all 7 bug tasks plus 20 safe tasks), whose harnesses are single-entry and well within monolithic BMC's reach. **Tier B (the frontier):** all 29 *bug* tasks of LDV-LINUX-3.4-SIMPLE (`unreach-call` ground truth *false*), Linux device-driver harnesses of 137–395 KB of CIL-preprocessed C spanning many functions, where monolithic BMC must encode the whole driver at once. Tier B is a *falsification* experiment: each task contains a reachable `reach_error`, and we measure how many each tool catches with a concrete counterexample.

b) Baselines: We compare BMC-Agent against the three established verifiers summarized below.

Verifier	Technique and configuration
CBMC	Bounded model checking (per-tier unwinding; see captions); BMC-Agent's default BMC backend.
ESBMC	SMT-based model checking with k-induction for unbounded proofs.
CPAchecker	Configurable program analysis with CEGAR-based predicate abstraction; SV-COMP configuration.

TABLE II: Tier A control: AWS-C-COMMON, 27-task sample (20 safe / 7 bug), `unreach-call` property, LP64 data model. CBMC unwind bound 64; ESBMC with k-induction. Counts: safe tasks correctly proved, false alarms (safe reported buggy), bugs caught, and unknown (timeout or out of memory). No tool reported a bug as safe.

Tool	Safe proved	False alarms	Bugs caught	Unknown
BMC-Agent	**18/20**	0	**7/7**	2
CBMC	**18/20**	0	**7/7**	2
ESBMC	13/20	**3**	5/7	6
CPAchecker	5/20	1	4/7	17

c) Tier A – BMC-Agent is sound and competitive where BMC is tractable: On AWS-C-COMMON, BMC-Agent proves $18/20$ safe tasks and catches all $7/7$ bugs with no false alarms, *tying* the strongest baseline: CBMC also scores $18/20$ and $7/7$, and the two leave the *identical* pair of safe tasks unknown (both SAT-capacity limits). On these single-entry harnesses BMC-Agent's orchestration inlines the full call closure, issuing effectively the same monolithic query as CBMC, so Tier A establishes that the agentic orchestration introduces no regression on code BMC already handles. ESBMC proves fewer safe tasks and raises 3 false alarms; CPAchecker leaves most tasks unknown (Table II).

TABLE III: Tier B frontier: LDV-LINUX-3.4-SIMPLE, all 29 bug tasks (`unreach-call` = *false*), LP64. A *catch* is a concrete `reach_error` counterexample; *unsound* = a reachable error reported safe; *unknown* = timeout or out-of-memory.

Tool	Catches	Unsound	Unknown
BMC-Agent	**18/29**	0	11
CPAchecker	13/29	0	16
ESBMC (k-induction)	10/29	**1**	18
CBMC ($k \leq 3$)	0/29	0	29

d) Tier B – BMC-Agent improves bug-finding where monolithic CBMC catches nothing: On the 29 LDV-LINUX bug tasks, monolithic CBMC catches 0, timing out or exhausting memory on essentially every driver even at the lowest unwinding bounds (k=1 to 3): it cannot encode the whole program. BMC-Agent's agentic orchestration shrinks the formula enough that the *same* CBMC engine catches $18/29$, with no unsound verdicts (every catch is a concrete `reach_error` counterexample). CPAchecker catches $13/29$ and ESBMC $10/29$; ESBMC additionally returns one unsound false-safe (megaraid_mm, whose reachable error its k-induction inductive step wrongly certifies unreachable). BMC-Agent thus catches the most bugs of any tool here and is the only one combining the top catch-count with zero unsound verdicts (Table III).

e) Complementarity: catching bugs the existing tools cannot find: The headline counts understate the picture: of BMC-Agent's 18 catches, 4 are caught by *neither* ESBMC nor CPAchecker, on drivers both leave *unknown*. BMC-Agent is thus a *complement* to these tools, not a replacement: it extends sound, witness-producing bug-finding to large multi-function drivers monolithic BMC cannot encode.

f) Why the remaining tasks are unknown: The generated harnesses for the 11 uncaught LDV tasks compile and convert cleanly. Their remaining *unknown* verdicts result from SAT-solver timeouts or memory exhaustion during driver analysis, the same limitations that leave monolithic CBMC with 0 detected bugs. This is precisely the scaling barrier that contract-based modular abstraction is designed to overcome.

C. RQ2: Comparison with FM-Agent

RQ2 compares BMC-Agent with FM-Agent [10], the closest prior LLM-based verification agent. We run both systems on the same VibeOS kernel corpus and manually review their outputs, counting a finding only when the reviewed source contains a concrete bug at a reachable VibeOS boundary; reports that require additional backend, environment, or helper-level assumptions are excluded. We do not use target reproduction as a comparison column, because the current FM-Agent artifact provides no equivalent target-replay layer.

a) Results: Table IV summarizes the comparison. FM-Agent reports at the function level: its 120 candidate mismatches yield 20 confirmed bugs in 20 distinct functions. BMC-Agent reports at the finer checked-property level: its 43 candidates yield 21 confirmed bugs in 13 distinct functions, so one function can contribute several confirmed bugs when independent properties fail. Table V breaks the confirmed findings down by bug class.

TABLE IV: RQ2: BMC-Agent vs. FM-Agent on the VibeOS kernel corpus. Raw candidates are each tool's native output unit; confirmed bugs are manually source-validated reports. Confirmed functions deduplicate those reports by function name.

Tool	Raw cand.	Conf. bugs	Conf./raw	Conf. funcs.
FM-Agent	120	20	17%	20
BMC-Agent	43	21	49%	13

The four bug classes are: *numeric/range*: integer overflow, truncation, or signed-shift undefined behaviour (e.g. `read32` and `fat32_init` reconstruct a 32-bit field with `p[3]<<24`, shifting a high byte into the sign bit); *object-bound/pointer*: out-of-bounds reads or writes and invalid pointer arithmetic (e.g. `elf_process_relocations` writes the relocated value past the mapped target); *API-contract/ordering*: a documented contract violated, such as a partial failure reported as success (`virtio_sound_play_pcm` returns `0` after a mid-buffer submit fails) or validation performed after an unsafe operation; and *protocol-length*: a packet length field used without a bounds check (`ip_handle` underflows the IPv4 payload length).

b) Findings: The two systems are complementary rather than one dominating the other. FM-Agent produces broader, specification-oriented reports, more evenly distributed across classes (notably the API-contract and ordering defects that BMC-Agent does not target). BMC-Agent produces fewer candidates but grounds each in a CBMC counterexample, concentrating on implementation-level memory-safety and undefined-behaviour defects (out-of-bounds accesses, invalid pointer arithmetic, integer overflow) at untrusted-input boundaries such as ELF, packet, and filesystem parsing, where unconstrained inputs are the realistic threat model and the bounded counterexample is a concrete witness of reachability. Their confirmed-function overlap is small but non-empty: `malloc`, `fb_draw_char`, `vfs_readdir`, and `ip_handle`.

D. RQ3: Specification Quality

BMC-Agent's specification synthesis is a pluggable component and deliberately naive relative to dedicated synthesizers; we therefore *calibrate* it against the closest specialized tools rather than aiming to surpass them, running BMC-Agent in a matching specification-emitting mode.

a) C/ACSL (AutoSpec): On the 251 C programs of the AutoSpec [11] benchmark, both systems use Claude Sonnet 4.6 to generate ACSL specifications, which are checked against the *same* Frama-C/WP proof obligations. We run five independent attempts per case. A case counts as solved when WP discharges all generated and benchmark obligations in at least one attempt, and the per-attempt success rate is the

TABLE V: Confirmed VibeOS findings by primary bug class. Entries give report-level confirmed bug counts and distinct function counts, followed by the function names where applicable. A function can appear in more than one row when different confirmed properties are in that function.

Class	FM-Agent	BMC-Agent
Numeric/range	**11 bugs / 11 funcs:** `fat32_file_size`, `fat32_init`, `fat32_read_file_offset`, `read32`, `do_request`, `vfs_read`, `mouse_set_pos`, `calloc`, `malloc`, `fb_draw_char`, `fb_fill_rect`.	**13 bugs / 9 funcs:** `elf_calc_size`, `elf_process_relocations`, `elf_load_at`, `fb_draw_char`, `fb_draw_string`, `malloc`, `dns_resolve`, `vfs_write`, `vfs_append`.
Object-bound/pointer	**2 bugs / 2 funcs:** `vfs_readdir`, `vfs_lookup`.	**7 bugs / 6 funcs:** `elf_process_relocations`, `elf_load_at`, `fb_draw_char`, `process_create`, `readdir_callback`, `vfs_readdir`.
API contract / ordering	**6 bugs / 6 funcs:** `virtio_sound_play_pcm`, `vfs_rename`, `vfs_mkdir`, `vfs_create`, `icmp_send_echo_request`, `kapi_delete_dir`.	**0**
Protocol / format	**1 bug / 1 func:** `ip_handle`.	**1 bug / 1 func:** `ip_handle`.

fraction of attempts that do. BMC-Agent proves 209/251, exceeding AutoSpec's published 199 but trailing a same-model AutoSpec rerun (219) by ten. The gap is concentrated in the Frama-C-problems, SV-COMP, and OOPSLA subsets, where AutoSpec's bottom-up loop-refinement procedure retries invariant inference when invariants are missing; on SyGuS/-Code2Inv, both systems solve 121 of the 133 cases, and BMC-Agent has the higher per-attempt success rate. A paired manual audit found the two comparable in the number of correct specifications, with different failure modes: AutoSpec more often omits a decisive relational invariant, BMC-Agent occasionally emits an overly strong or malformed ACSL clause.

b) Java/JML (SpecGen): On SpecGenBench (N=120) and the SV-COMP Java subset (N=265), the original SpecGen [20] pipeline and a configuration of BMC-Agent that generates JML annotations both use Claude 4.6 and GPT-3.5. OpenJML checks the generated annotations, with ten independent attempts per case (a case is solved when at least one attempt discharges the OpenJML obligations). SpecGen's verifier-guided JML refinement keeps it ahead on case coverage (e.g. 116 vs. 112 of 120 on SpecGenBench with Claude); however BMC-Agent has more successful attempts in all four matched settings, i.e. it is competitive and more stable as a JML annotation generator.

E. RQ4: Findings on Real-World Software

Applied to mature and heavily-fuzzed real-world software, BMC-Agent surfaced **68 source-audited defects across eleven software projects** (Table VI). Each was confirmed by human source audit rather than by the triage agent alone. For C/C++ targets, confirmations generally include standalone sanitizer reproducers; for Rust CCC, confirmations use Kani counterexamples and source audit. At least two defects were filed and accepted as upstream security advisories; the remainder are under coordinated disclosure. Advisory identifiers are withheld for double-blind review, but the artifacts are available as the supplementary materials.

a) The headline cluster: u-boot: The largest cluster, 32 defects, is in u-boot, onboarded to OSS-Fuzz only in 2026, whose squashfs, btrfs, ext4, and EFI image parsers remain under-fuzzed; several are reachable only when the bootloader mounts attacker-supplied removable media, for which no fuzz harness exists yet. Nearly all belong to a single class, an on-disk length or count trusted without bounding it against the buffer, which is the unenforced-contract pattern of the motivating example (§II), found here compositionally across loader call chains.

TABLE VI: Confirmed real-world defects after human source audit. C/C++ entries are memory-safety defects, while the Rust CCC entries are arithmetic/panic-class correctness defects confirmed by Kani counterexamples and source audit. The largest cluster is u-boot, a freshly-onboarded and under-fuzzed OSS-Fuzz target.

Project	Domain	Defects
OSS-Fuzz-integrated projects		
u-boot	bootloader: filesystem / EFI parsers	32
gpac	multimedia framework	10
libbej	Redfish BEJ decoder	4
jq	JSON processor (filed upstream)	2
libredwg	DWG/CAD reader	2
libarchive	archive formats	1
Other open-source (not in OSS-Fuzz)		
libmikmod	module-audio loaders	4
libmodplug	module-audio loaders	2
adplug	AdLib-audio loaders	3
CCC	AI-generated Rust C compiler	7
AWS Neuron	Trainium/Inferentia ioctl handler	1
11 projects		**68**

b) Confirmed-real vs. unreported: Sanitizer-confirmed-real is a stronger bar than previously-unreported, and we separate the two. All remain unfixed upstream; for novelty (no prior public report) the evidence is strongest for the u-boot cluster and for the libmikmod defects, a rodata and intra-array over-read class that AddressSanitizer-based fuzzing structurally cannot catch. We treat several gpac and libmodplug findings as possibly already known, pending per-bug triage. The figure we claim is therefore confirmed-real-and-unfixed, not all-unreported.

c) Clean verifications as positive evidence.: Beyond defects, many functions on these hardened surfaces completed the entry-spec, BMC, classification, and realism phases without surfacing any REALISTIC finding (for example, OpenSSL's ASN.1 DER decoders and libxml2's XPath pattern compiler).

On code this heavily fuzzed, a clean, spec-relative sweep is itself a result: bounded verification evidence that the analysed property holds to the unwinding bound, which an LLM-only auditor cannot provide.

F. Threats to Validity

a) Soundness of agent decisions: An overly weak specification or an overly aggressive abstraction can silently miss a defect, the one failure mode the pipeline cannot self-detect; we bound it by keeping inlining promote-only over a structural error-reaching floor, though a permissive yet consistent spec leaves no trace.

b) Over-approximation and bounded soundness: Stubs and frame-havoc over-approximate, so a witness may be spurious; BMC-Agent validates every reported witness against real code (callee feasibility, reachability, dynamic replay), whose signal labels are host-side evidence rather than target-platform claims. A clean verdict holds only up to the unwinding bound.

c) Evaluation: Lacking ground truth for LLM-generated code, true positives are judged by a hybrid criterion that combines a generated reproducer, compiled and run to trigger the defect where one can be produced, with manual source review of every confirmed finding; the SV-COMP tasks are instead scored directly against the given labels.

d) External validity: Our subjects span three deliberately different populations, namely LLM-generated systems code, mature OSS-Fuzz-hardened libraries, and curated verification benchmarks, chosen to probe generality. Even so, the absolute defect rates and catch counts depend on corpus selection and may not transfer to other domains, languages, or systems.

V. Related Work

a) LLM-based specification and invariant synthesis: Inferring program properties predates LLMs: Daikon mines likely invariants from executions [21], Houdini filters candidate annotations for ESC/Java [22], and others mine specifications from how code is *used*, including API-usage patterns from execution traces [23], preconditions that hold at a procedure's call sites [24], and temporal specifications from object usage [25]. Recent work revisits this goal with LLMs. AutoSpec [11] and SpecGen [20] synthesize ACSL/JML contracts for deductive verifiers in a generate–check–refine loop; Faria et al. [26] add test oracles and verification annotations, and separation-logic specifications have also been synthesized [27]. SpecSyn [28] and C2S [29] infer formal specifications from code and natural-language comments; further studies assess how reliably LLMs produce and reason about such specifications [30], [31]. A parallel thread targets a single artifact: Loopy [32] proposes and repairs loop invariants against a checker, Wu et al. [33] and Pirzada et al. [34] pair LLM-generated invariants directly with bounded model checking, and AutoVerus [35] generates the proof annotations Verus needs for Rust, while Safe4U [36] flags unsound safe encapsulations of unsafe code; the premise that verification is gated on specifications is itself long-studied [37]. These systems produce specifications, invariants, or proofs for a downstream tool; BMC-Agent instead treats specification inference as one stage of a verification workflow, integrating caller-grounded contracts with bounded model checking, counterexample validation, and compositional refinement.

b) Agentic verification and LLM bug finding: Several systems pair LLM agents with formal reasoning. FM-Agent [10] infers Hoare-style contracts and checks them with LLM reasoning; Lemur [12] has LLMs synthesize artifacts checked by external engines; ConVer [38] scales deductive verification from LLM-generated contracts given a user-provided property; and Code-Augur [39] and specification-guided detection [40] expose security assumptions as assertions and flag violating code. FM-Agent and Code-Augur are closest to BMC-Agent (all infer and refine specifications) but differ in the *verification oracle*: FM-Agent relies on LLM reasoning, Code-Augur on a guided fuzzer, and LLM-powered symbolic execution [41] on an approximate engine, whereas BMC-Agent discharges each specification with bounded model checking, exhaustively covering inputs up to a bound and yielding concrete counterexamples and bounded clean verdicts. A related line uses LLMs for interactive proof construction [42]–[45] and for bug and vulnerability detection [46], which yield proofs or bug reports rather than the model-checked verdicts BMC-Agent produces.

c) Model checking, program analysis, and deductive verification: Bounded model checkers [5]–[7], [14] verify soundly relative to a supplied specification, and frameworks such as SeaHorn [47], CPAchecker [15], and BLAST [48] scale verification via abstraction, Horn-clause reasoning, and CEGAR. SLAM/SDV checks fixed driver-API properties [49], Smatch [50] performs pattern-based kernel analysis, and Infer [51] scales compositional bug finding through procedure summaries. Deductive verifiers such as Boogie [52], Dafny [53], Why3 [54], Frama-C/WP [55], and Verus [56] give strong guarantees but require substantial user-supplied annotations. These systems excel once properties exist; BMC-Agent targets the complementary problem of inferring and refining the specifications that make them applicable to unannotated code, while retaining the concrete witnesses produced by bounded model checking.

d) Symbolic execution, compositional verification, and CEGAR: KLEE [57] and EXE [58], concolic engines (DART [59], CUTE [60]), and hybrid symbolic–fuzzing techniques [61]–[63] expose bugs through path exploration, and large-scale fuzzing infrastructure such as OSS-Fuzz [18] and AFL++ [64] has hardened many libraries we evaluate; unlike BMC-Agent, these generate inputs rather than per-function specifications or bounded clean verdicts. Serval [65] symbolically evaluates systems software within decidable fragments. BMC-Agent builds on assume-guarantee reasoning [8] and CEGAR [9]: function contracts act as assume-guarantee obligations, and unrealistic counterexamples trigger refinement at the specification level, propagated compositionally across the call graph and persisted across runs.

VI. Conclusion

We have presented *agentic model checking* and BMC-Agent, which place LLM agents around a bounded model checker under the principle that *agents propose, solvers verify*: the agents take on the semantic judgment a checker lacks, while soundness stays with a deterministic backend. The central idea is to let an agent *orchestrate* the checker rather than fix a single recipe, driving one BMC engine to different operating points so it scales to large, multi-function code that a monolithic encoding cannot solve. Across LLM-generated systems code, verification benchmarks, and hardened libraries, BMC-Agent finds confirmed defects in heavily tested code, including bugs in real drivers beyond the reach of monolithic model checking. It also produces bounded clean verifications. More broadly, the results suggest that LLM agents are most useful in verification not as a replacement for the solver but as the *orchestrator* that decides how to apply it.

References

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde De Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[2] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of Github Copilot's code contributions," *Communications of the ACM*, vol. 68, no. 2, pp. 96–105, 2025.

[3] S.-C. Dai, J. Xu, and G. Tao, "Rethinking the evaluation of secure code generation," in *International Conference on Software Engineering*. ACM, 2026, pp. 336–348.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1999, pp. 193–207.

[5] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2004, pp. 168–176.

[6] A. VanHattum, D. Schwartz-Narbonne, N. Chong, and A. Sampson, "Verifying dynamic trait objects in Rust," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 2022, pp. 321–330.

[7] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik, "JBMC: A bounded model checking tool for verifying Java bytecode," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 183–190.

[8] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in C," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 388–402, 2004.

[9] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *International Conference on Computer Aided Verification*. Springer, 2000, pp. 154–169.

[10] H. Ding, Z. Wang, and H. Chen, "FM-Agent: Scaling formal methods to large systems via LLM-based Hoare-style reasoning," *arXiv preprint arXiv:2604.11556*, 2026.

[11] C. Wen, J. Cao, J. Su, Z. Xu, S. Qin, M. He, H. Li, S.-C. Cheung, and C. Tian, "Enchanting program specification synthesis by large language models using static analysis and program verification," in *International Conference on Computer Aided Verification*. Springer, 2024, pp. 302–328.

[12] H. Wu, C. Barrett, and N. Narodytska, "Lemur: Integrating large language models in automated program verification," in *International Conference on Learning Representations*, 2024.

[13] R. Liu, M. Chen, L.-I. Wu, J. Ke, and G. Li, "Enhancing automated loop invariant generation for complex programs with large language models," *Science of Computer Programming*, vol. 248, p. 103387, 2026. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642325001261

[14] L. Cordeiro, B. Fischer, and J. Marques-Silva, "SMT-based bounded model checking for embedded ANSI-C software," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 957–974, 2012.

[15] D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 184–190.

[16] "Vibeos," https://github.com/kaansenol5/VibeOS, 2026, GitHub repository, accessed 2026-05-18.

[17] "Claude's C compiler (CCC)," https://github.com/anthropics/claudes-c-compiler, 2026, GitHub repository.

[18] O. Chang, J. Metzman, M. Moroz, M. Barbella, and A. Arya, "OSS-Fuzz: Continuous fuzzing for open source software," URL: https://github.com/google/oss-fuzz, 2016.

[19] D. Beyer and J. Strejček, "Evaluating software verifiers for C, Java, and SV-LIB: (report on SV-COMP 2026)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2026, pp. 461–502.

[20] L. Ma, S. Liu, Y. Li, X. Xie, and L. Bu, "SpecGen: Automated generation of formal program specifications via large language models," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2025, pp. 16–28.

[21] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[22] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java," in *International Symposium of Formal Methods Europe*. Springer, 2001, pp. 500–517.

[23] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 371–382.

[24] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Static specification inference using predicate mining," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 123–134, 2007.

[25] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," *Automated Software Engineering*, vol. 18, no. 3, pp. 263–292, 2011.

[26] J. P. Faria, E. Trigo, V. Honorato, and R. Abreu, "Automatic generation of formal specification and verification annotations using LLMs and test oracles," *arXiv preprint arXiv:2601.12845*, 2026.

[27] T. Suresh, D. Korczynski, and J. Vanegue, "Agentic separation logic specification synthesis," *arXiv preprint arXiv:2605.27531*, 2026.

[28] L. Ma, S. Liu, Y. Li, Q. Wu, H. Wang, and L. Bu, "SpecSyn: LLM-based synthesis and refinement of formal specifications for real-world program verification," *arXiv preprint arXiv:2604.21570*, 2026.

[29] J. Zhai, Y. Shi, M. Pan, G. Zhou, Y. Liu, C. Fang, S. Ma, L. Tan, and X. Zhang, "C2S: translating natural language comments to formal program specifications," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 25–37.

[30] T. Le-Cong, B. Le, and T. Murray, "Can LLMs reason about program semantics? A comprehensive evaluation of LLMs on formal specification inference," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2025, pp. 21 991–22 014.

[31] D. S. Bouras, Y. Dai, T. Wang, Y. Xiong, and S. Mechtaev, "Hoare-Prompt: Structural reasoning about program correctness in natural language," in *International Conference on Software Engineering*. ACM, 2026, pp. 246–258.

[32] A. Kamath, N. Mohammed, A. Senthilnathan, S. Chakraborty, P. Deligiannis, S. K. Lahiri, A. Lal, A. Rastogi, S. Roy, and R. Sharma, "Leveraging LLMs for program verification," in *2024 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2024, pp. 107–118.

[33] G. Wu, W. Cao, Y. Yao, H. Wei, T. Chen, and X. Ma, "LLM meets bounded model checking: Neuro-symbolic loop invariant inference," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2024, pp. 406–417.

[34] M. A. Pirzada, G. Reger, A. Bhayat, and L. C. Cordeiro, "LLM-generated invariants for bounded model checking without loop unrolling," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2024, pp. 1395–1407.

[35] C. Yang, X. Li, M. R. H. Misu, J. Yao, W. Cui, Y. Gong, C. Hawblitzel, S. Lahiri, J. R. Lorch, S. Lu *et al.*, "AutoVerus: Automated proof generation for Rust code," *Proceedings of the ACM on Programming Languages*, vol. 9, no. OOPSLA2, pp. 3454–3482, 2025.

[36] H. Li, B. Wang, X. Hu, and X. Xia, "Safe4U: Identifying unsound safe encapsulations of unsafe calls in Rust using LLMs," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 457–480, 2025.

[37] D. Neider and R. Roy, "What is formal verification without specifications? A survey on mining LTL specifications," in *Principles of Verification: Cycling the Probabilistic Landscape: Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part III*. Springer, 2024, pp. 109–125.

[38] M. A. Pirzada, W. Wang, Y. Charalambous, K. Korovin, and L. C. Cordeiro, "ConVer: Using contracts and loop invariant synthesis for scalable formal software verification," *arXiv preprint arXiv:2605.27051*, 2026.

[39] Z. Luo, M. Zafar, D. Wolff, and A. Roychoudhury, "Code-Augur: Agentic vulnerability detection via specification inference," *arXiv preprint arXiv:2606.18619*, 2026.

[40] H. Zhu, J. Li, C. Gao, J. Qian, Y. Dong, H. Liu, L. Wang, Z. Wang, X. Hu, and G. Li, "Specification-guided vulnerability detection with large language models," *arXiv preprint arXiv:2511.04014*, 2025.

[41] Y. Li, R. Meng, and G. J. Duck, "Large language model powered symbolic execution," *Proceedings of the ACM on Programming Languages*, vol. 9, no. OOPSLA2, pp. 3148–3176, 2025.

[42] S. Polu and I. Sutskever, "Generative language modeling for automated theorem proving," *arXiv preprint arXiv:2009.03393*, 2020.

[43] E. First, M. N. Rabe, T. Ringer, and Y. Brun, "Baldur: Whole-proof generation and repair with large language models," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2023, pp. 1229–1241.

[44] K. Yang, A. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. J. Prenger, and A. Anandkumar, "LeanDojo: Theorem proving with retrieval-augmented language models," *Advances in Neural Information Processing Systems*, vol. 36, pp. 21 573–21 612, 2023.

[45] H. Tu, H. Zhao, Y. Song, M. Zafar, R. Meng, and A. Roychoudhury, "Agentic verification of software systems," *Proc. ACM Softw. Eng.*, no. FSE, pp. 3558–3581, 2026.

[46] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, and Y. Chen, "Vulnerability detection with code language models: How far are we?" in *International Conference on Software Engineering*. IEEE, 2025, pp. 1729–1741.

[47] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The SeaHorn verification framework," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 343–361.

[48] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002, pp. 58–70.

[49] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, 2001, pp. 203–213.

[50] D. Carpenter, "Smatch: Pluggable static analysis for C," https://lwn.net/Articles/691882/, 2016.

[51] C. Calcagno and D. Distefano, "Infer: An automatic program verifier for memory safety of C programs," in *NASA Formal Methods Symposium*. Springer, 2011, pp. 459–465.

[52] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *International Symposium on Formal Methods for Components and Objects*. Springer, 2005, pp. 364–387.

[53] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2010, pp. 348–370.

[54] J.-C. Filliâtre and A. Paskevich, "Why3—where programs meet provers," in *European Symposium on Programming*. Springer, 2013, pp. 125–128.

[55] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: a software analysis perspective," in *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, ser. SEFM'12. Springer, 2012, p. 233–247. [Online]. Available: https://doi.org/10.1007/978-3-642-33826-7_16

[56] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, "Verus: Verifying Rust programs using linear ghost types," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 286–315, 2023.

[57] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*. USENIX Association, Dec. 2008, pp. 209–224. [Online]. Available: https://www.usenix.org/legacy/event/osdi08/tech/full_papers/cadar/cadar.pdf

[58] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, pp. 1–38, 2008.

[59] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 213–223.

[60] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[61] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *NDSS*, vol. 8. The Internet Society, 2008, pp. 151–166.

[62] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 380–394.

[63] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!" in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 181–198.

[64] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[65] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang, "Scaling symbolic evaluation for automated verification of systems code with Serval," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. ACM, 2019, p. 225–242. [Online]. Available: https://doi.org/10.1145/3341301.3359641