Small Harnesses, Default Checks, Limited Coverage: Bounded Model Checking in Practice

Kevin Kolyakov
University of Waterloo
Waterloo, Canada
kkolyakov@uwaterloo.ca

Ricardo Calvo
Purdue University
West Lafayette, USA
rcalvome@purdue.edu

James C. Davis
Purdue University
West Lafayette, USA
davisjam@purdue.edu

Paschal C. Amusuo
Purdue University
West Lafayette, USA
pamusuo@purdue.edu

***Abstract*—The growing use of AI to generate software and verification artifacts has increased the need for automated systems that provide reliable software assurance. Bounded model checking (BMC) offers a practical foundation for such systems by automatically checking correctness and security properties directly on software. However, effective verification requires choices about properties, input assumptions, function models, and execution bounds that influence the guarantees obtained. Little systematic evidence exists on how practitioners make these choices, limiting the empirical foundation for developing automated verification systems.**

We present an empirical study of BMC practices in open-source C and Rust projects using CBMC and Kani. Through LLM-assisted open coding, static analysis, and artifact execution, we characterize verification properties, artifact complexity, and verification outcomes. Our findings reveal diverse verification objectives and generally small harnesses, but also property-specification errors and extensive use of execution bounds and environment models. Most artifacts complete verification quickly and achieve high coverage within their scope, while repository-level coverage decreases with project size. Our findings and annotated dataset provide empirical baselines for verification automation and motivate improved techniques for property validation, harness generation, and scaling verification breadth.

***Index Terms*—Bounded Model Checking, Formal Verification, Software Assurance**

I. Introduction

Rapid AI-assisted code generation and advances in AI-assisted formal verification are increasing demand for automated systems that establish software correctness and security [1], [2]. Such systems must keep pace with software production while uncovering subtle defects that threaten reliability and security. However, their effectiveness depends on choices about which properties to verify, how to model software behavior and its environment, and how to configure verification tools. Understanding how practitioners make these choices is therefore important for designing automated verification systems that provide meaningful software assurance.

Bounded model checking (BMC) [3] is particularly suited to such automation because it verifies software properties with limited manual proof guidance. Unlike techniques requiring separately constructed system models or interactive proofs, BMC tools operate directly on existing software and automatically check built-in safety properties alongside custom specifications. Prior studies have demonstrated BMC's effectiveness in uncovering high-impact vulnerabilities [4] and its practicality in industrial software development [5], [6]. However, little empirical evidence exists on the properties and other verification choices practitioners make, leaving researchers without a clear empirical basis for automating these decisions.

This paper addresses this gap through an empirical study of BMC practices in open-source software projects. We analyze 2,005 verification artifacts from 24 C and Rust projects using two popular BMC tools, CBMC [7] and Kani [8]. We employ LLM-assisted open coding to characterize the properties practitioners specify and static analysis to measure the complexity and configurations of their verification artifacts. We also execute these artifacts to evaluate their verification results, execution costs, and code coverage.

Our study produces several findings that inform the design and evaluation of automated verification systems. We identify 12 categories of custom verification properties and specification errors in 35 harnesses across ten projects, highlighting the need for property validation. Although harnesses are generally small (median seven lines), 63% specify input preconditions, 54% configure loop bounds, and 31.2% use function models, all of which influence verification scope. Most artifacts (97.2%) complete within resource limits with high artifact-level coverage (median 98%), but project-level coverage decreases with project size, providing practical baselines for automation while exposing challenges in verification breadth.

In summary, we make three contributions:

- A taxonomy of verification properties and empirical measurements of their adoption across BMC projects.
- An empirical characterization of the complexity, execution costs, and verification outcomes of BMC artifacts.
- A curated dataset of 2,005 BMC artifacts from 24 projects, providing a foundation for future empirical studies and evaluation of automated verification techniques.

Significance: Our study advances the development of automated verification systems by providing empirical evidence of existing BMC practices and their limitations. The findings and dataset provide practical guidance and evaluation baselines for property inference, verification target selection, and automated harness generation. They also motivate techniques for validating generated properties and assumptions, ensuring that successful verification establishes meaningful guarantees.

These advances are increasingly important as software engineers and AI agents produce code at greater speeds, creating demand for scalable and trustworthy software assurance.

II. Background and Related Work

This section introduces the theoretical and practical foundations for our work.

A. Bounded Model Checking (BMC)

1) As a Verification Technique: Bounded model checking (BMC) [3] is an automated verification technique that checks software properties within bounded program executions. It encodes program behaviors and the negation of target properties into a propositional formula and uses constraint solving to check whether the target properties can be satisfied.

$$\Phi_{b,M}(\tau) = E_{b,M}(\tau) \wedge \neg P(\tau), \tag{1}$$

Given a program whose executions are encoded by $E_{b,M}(\tau)$ under execution bounds b and environment model M, BMC checks whether any execution sequence τ violates the target property $P(\tau)$ by testing the satisfiability of $\Phi_{b,M}(\tau)$. A satisfying assignment produces a counterexample indicating a possible property violation, while unsatisfiability establishes that the property holds for all modeled executions.

To control verification complexity, practitioners can bound loop iterations and recursion depth (b) or replace callers or dependent functions with simpler behavioral models (M). However, insufficient bounds and overly restrictive models can exclude executions containing real defects, while overly permissive models may introduce spurious violations [4], [9]. Thus, verification guarantees depend on the properties, bounds, and environment models selected.

2) Tools: Several BMC tools support mainstream programming languages, including CBMC [7] and ESBMC [10] for C, JBMC [11] for Java, and Kani [12] for Rust. These tools translate source programs into verification conditions for satisfiability solvers and automatically check common safety properties, including memory safety, arithmetic overflow, and assertion violations. Practitioners can also specify custom properties through assertions and function contracts and configure which built-in checks are enabled.

Compared to other verification tools, BMC offers a tradeoff between verification effort and the scope of its guarantees. Unlike deductive verifiers [13], [14] and interactive theorem provers [15]–[17], BMC does not require separately-constructed formal system models, makes function and loop invariants optional, and does not require manual proof guidance. However, BMC only supports properties in propositional and first order logic and its guarantees is limited by execution bounds and environment assumptions.

B. Component-Level BMC and Harnesses

To manage verification complexity, practitioners often verify individual functions or groups of functions rather than entire systems. This component-level approach resembles unit testing and has been termed unit proofing [4]. Practitioners construct verification harnesses that initialize symbolic inputs, constrain their values through assumptions, invoke target functions, and assert expected properties. §II-B illustrates these steps using CBMC and Kani harnesses for an increment function.

(a) CBMC (C)

```
1  // Function contract
2  unsigned inc(unsigned x)
3    __CPROVER_requires(x < 100)
4    __CPROVER_ensures(
5      __CPROVER_return_value
6        == x + 1)
7  { return x + 1; }
8
9  // Self-contained harness
10 void check_inc(void) {
11   unsigned x = nondet_uint();
12   __CPROVER_assume(x < 100);
13   unsigned y = inc(x);
14   __CPROVER_assert(
15     y == x + 1, "inc");
16 }
17
18 // Contract-checking harness
19 void contract_harness(void) {
20   inc(nondet_uint());
21 }
22 // --enforce-contract inc
```

(b) Kani (Rust)

```
1  // Function contract
2  #[kani::requires(x < 100)]
3  #[kani::ensures(
4    |r: &u32| *r == x + 1)]
5  fn inc(x: u32) -> u32 { x + 1 }
6
7
8  // Self-contained harness
9  #[kani::proof]
10 fn check_inc() {
11   let x: u32 = kani::any();
12   kani::assume(x < 100);
13   let y = inc(x);
14   assert!(y == x + 1);
15 }
16
17 // Contract-checking harness
18 #[kani::proof_for_contract(
19   inc)]
20 fn contract_harness() {
21   inc(kani::any());
22 }
```

Listing 1: CBMC (left) and Kani (right) examples of self-contained and contract-based verification.

In the self-contained harnesses in §II-B, the CBMC and Kani examples create symbolic inputs (line 11), restrict them to values below 100 (line 12), and invoke `inc()` before asserting that its result equals the input plus one. These assumptions define the verification domain, while the assertions specify the properties being verified. The listing also illustrates function contracts (CBMC lines 2–7; Kani lines 2–5), which express the same requirements as preconditions and postconditions. The contract-checking harnesses invoke `inc()` to verify these requirements using CBMC's contract-enforcement option or Kani's `proof_for_contract` annotation.

Beyond input models and properties, harnesses may replace dependent functions with custom models and configure global or loop-specific unwinding bounds. Verified function contracts can also abstract function implementations when verifying their callers, while loop contracts support invariant-based reasoning without relying solely on finite unwinding [12]. Together with build scripts and tool configurations, these constructs form the verification artifacts that determine the scope, guarantees, and cost of each verification task.

C. Related Work on BMC

We distinguish prior work improving and evaluating BMC from studies of verification practices in software projects.

1) BMC Techniques and Applications: Prior research has focused on improving BMC techniques, evaluating their effectiveness, and documenting their applications in specific settings. For example, industrial studies have demonstrated BMC's practicality in software development [6], [18], while other studies have evaluated its effectiveness in vulnerability discovery [4] and improvements to verification performance [19]. Recent research has also explored automating verification-task extraction [20] and specification generation [21]. Although these studies demonstrate BMC's capabil-

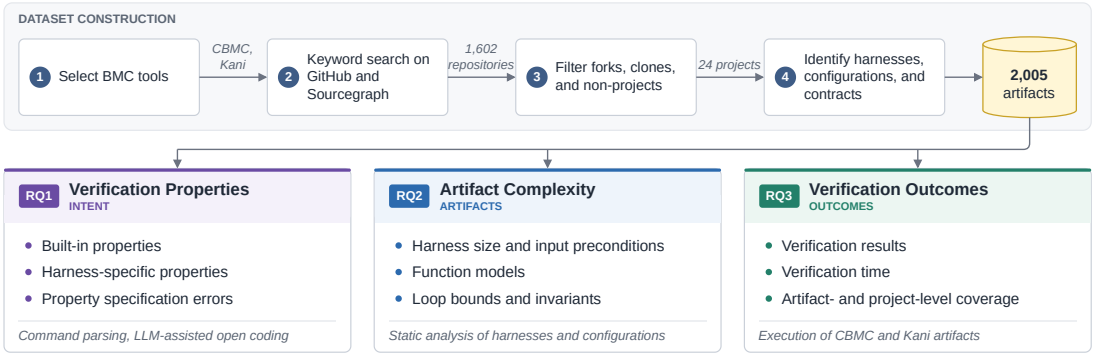


Fig. 1: Overview of our methodology. We select projects that contain CBMC and Kani artifacts, identify harnesses within these projects, and study their verification properties, artifact complexity and verification outcomes.

ities and opportunities for improvement, they provide limited evidence of BMC use across independent software projects.

2) Empirical Studies of Verification Practices: Another line of research empirically investigates verification artifacts and practices in existing software projects. For example, Matichuk *et al.* [22] studied relationships between specification and proof sizes in Isabelle, while Estler *et al.* [23] and Dietrich *et al.* [24] examined software contract usage across projects. Other studies have investigated program characteristics associated with verification-tool performance [25]. However, these studies do not cover the verification choices and artifacts relevant to BMC across software projects. Our study bridges these lines of research by providing empirical evidence of existing BMC practices to guide the development and evaluation of automated verification systems.

III. Knowledge Gap and Research Questions

Developing effective automated BMC systems requires making verification choices that establish meaningful software correctness and security guarantees. Studying the choices practitioners make can reveal the capabilities and limitations of existing practices and provide an empirical basis for improving verification automation. Yet, no prior study has systematically examined these practices across diverse software projects.

We address this gap through three research questions:

- **RQ1 (Properties):** What do open-source projects verify?
- **RQ2 (Artifact Complexity):** What are the characteristics and complexity of verification artifacts used in practice?
- **RQ3 (Verification Outcomes):** What verification results, execution costs, and code coverage do artifacts achieve?

Answering these questions can guide automated BMC systems that select meaningful properties, choose effective verification options, and produce understandable guarantees.

IV. Study Methodology

Figure 1 illustrates our methodology. We describe our dataset construction and research methods.

A. Dataset Construction

We study BMC artifacts in open-source software projects. We construct our dataset in four steps.

Step 1 — Selection of BMC Tools: We focus on two systems-programming ecosystems, C and Rust. We identified candidate BMC tools through academic publications, tool documentation, and software verification competitions [26], [27]. We shortlisted seven tools and conducted a preliminary GitHub search using tool-specific keywords and code annotations. Our search identified more repositories for Kani (96) and CBMC (38) than for ESBMC (10), Crux-MIR (8), SeaBMC (4), SMACK (2), and CPAchecker (1). Hence, we selected CBMC and Kani based on their relatively widespread use and mature industry support. Although preliminary, these results indicated their adoption in open-source projects.

Step 2 — Identification of BMC-based Projects: We conducted a more extensive search on GitHub and Sourcegraph to identify projects using the selected tools. For CBMC, our keywords included `__CPROVER_assume`, `cbmc-proof.txt`, and CBMC build configurations. For Kani, they included `#[kani::proof]`, `#[kani::proof_for_contract]`, and Kani invocation scripts. These searches yielded 1,602 candidate repositories.

Step 3 — Project Filtering and Selection: We applied two filtering criteria to identify independent software projects containing BMC artifacts. First, we excluded forks and clones using automated comparisons of repository names, proof filenames, and contents. Second, we used LLM-assisted screening to exclude tutorials, benchmarks, and non-software repositories. We manually audited a random sample of exclusions to assess the filtering accuracy.

Of the 457 remaining projects, we selected 12 per tool, prioritizing popularity (GitHub stars), maturity (age and maintainer counts), and application-domain diversity. Table I lists selected projects. We provide the list of eligible and selected projects and their attributes in the supplementary materials.

Step 4 — Identification of Verification Artifacts: We inspected each selected project to understand how its verification harnesses were organized and executed. We

TABLE I: Projects in this study, with their domain, size, GitHub stars (fetched 2026-10-27), and artifact count. Size counts non-blank, non-comment lines of code (cloc) at the studied commit, excluding data/markup files and vendored dependencies.

CBMC (12 projects, 1,320 artifacts)					Kani (12 projects, 685 artifacts)				
Project	**Domain**	**Size (KLOC)**	**# Stars**	**# Artifacts**	**Project**	**Domain**	**Size (KLOC)**	**# Stars**	**# Artifacts**
mldsa-native	Post-quantum signatures	173.6	116	218	materialize	Streaming SQL database	1,089.4	6,391	6
s2n-tls	TLS and cryptography	156.5	4,783	154	xln	Layer-2 payment network	937.8	102	28
FreeRTOS-Plus-TCP	Embedded networking	151.1	216	99	nonos-microkernel	Microkernel OS	517.3	300	108
mlkem-native	Post-quantum key encapsulation	116.2	254	190	zerocopy	Zero-copy memory conversion	250.2	2,625	61
FreeRTOS-Kernel	Real-time OS kernel	115.1	7,933	51	s2n-quic	QUIC transport protocol	233.6	1,377	114
aws-c-common	C runtime library	56.7	303	173	templar-contracts	DeFi lending contracts	225.5	11	46
aws-encryption-sdk-c	Client-side encryption	53.4	64	100	firecracker	MicroVM monitor	114.3	37,662	35
coreMQTT	MQTT messaging client	33.6	234	101	merc	Model-checking toolset	98.6	4	23
FreeRTOS-Cellular-Interface	Cellular modem interface	26.2	101	63	rust-bitcoin	Bitcoin protocol library	73.3	2,691	5
corePKCS11	PKCS#11 cryptographic API	17.2	46	30	spacewasm	WebAssembly interpreter	71.1	1,612	72
ota-for-aws-iot-embedded-sdk	IoT over-the-air updates	14.9	66	121	hifitime	Astrodynamics timekeeping	14.3	556	153
coreMQTT-Agent	Thread-safe MQTT agent	5.0	46	20	x86_64	x86-64 hardware abstractions	9.1	942	34

then developed a script to identify harness entry points, source files, configurations, and associated function contracts. We treated each distinct combination of these elements as one verification artifact. For each artifact, we recorded its verification target, source location, and whether it was self-contained or contract-based (§II-B). This process identified 2,005 artifacts across 24 projects, summarized in Table I.

To assess whether our dataset reflects human-developed BMC practices, we examined artifact commit histories for evidence of AI assistance. Only three projects (`zerocopy`, `spacewasm`, and `templar-contracts`) showed such evidence, covering 20–90% of their respective artifacts but only 5.2% of the full dataset. These findings suggest that our dataset largely reflects human practices while capturing emerging AI adoption, although unattributed AI use remain undetected.

B. RQ1: Verification Properties

Automated BMC systems must select properties that establish meaningful correctness and security guarantees. RQ1 investigates the properties practitioners verify to inform property inference and identify potential gaps in existing practices. We distinguish built-in properties configured through tools from custom properties in harnesses or function contracts.

1) Built-in Properties: We reviewed CBMC and Kani's interfaces to identify their supported built-in properties, default settings, and configuration options. CBMC (v6.x.x) supports 17 properties and Kani supports 11, with eight enabled by default in each tool. We grouped them into six categories: memory safety (*e.g.,* bounds checking), memory resource management (*e.g.,* memory leaks), arithmetic safety (*e.g.,* integer overflow), value validity (*e.g.,* enum range checking), runtime safety (*e.g.,* Rust panics), and unwinding assertions.

We developed an automated script to parse each artifact's execution commands and identify enabled properties, accounting for properties enabled by default. For each category, we measured the proportion of projects enabling some or all properties in some or all of their harnesses.

2) Harness-Specific Properties: We employed LLM-assisted open coding to identify and categorize custom properties specified in verification harnesses and function contracts. First, one author manually examined harnesses from two CBMC projects and one Kani project, identifying the first two property categories. The author then used GPT-6.1-sol through Codex to analyze the remaining harnesses in batches of four projects. The model assigned unambiguous properties to existing categories and flagged new or ambiguous cases for human review. The author reviewed these cases and introduced or refined categories as needed, retaining responsibility for the taxonomy and its definitions. While analyzing the harnesses, the model was also tasked to identify and report property specifications whose expression may differ from their intended meaning. The author reviewed these cases, categorized them and recorded them as property-specification errors.

To assess coding consistency, we selected one CBMC (ota-for-aws-iot-embedded-sdk) and one Kani (spacewasm) project with the greatest diversity of property categories. An independent GPT-6.1-sol agent classified properties in their harnesses using the resulting taxonomy. We measured agreement across 193 harnesses and 12 categories, treating each harness-

category pair as a binary decision. The agents agreed on 2,250 of 2,316 decisions (97.15%; Cohen's $\kappa = 0.823$), almost perfect by Landis and Koch's benchmarks [28].

C. RQ2: Artifact Complexity

Automated BMC systems must construct verification artifacts with appropriate input assumptions, function models, and execution bounds. RQ2 examines the complexity of these artifacts to inform their automated generation and identify opportunities to reduce verification effort. Building on prior work that uses proof size and the presence of proof constructs to approximate complexity [22], we analyze complexity along three dimensions: harness and input models, function models, and loop bounds. We use automated scripts to analyze all 2005 artifacts and extract these measurements.

1) Harness and Input Models: We measure harness size and the number of preconditions specified. For each artifact, we count the lines of code in its verification entry point, excluding function signatures, blank lines, and comments. We identify preconditions through CBMC's `__CPROVER_assume`, Kani's `kani::assume` and `kani::any_where`, and `requires` clauses in verified function contracts. Each invocation or contract clause counts as one precondition, regardless of the number of conditions it contains.

2) Function Models: We measure the prevalence and size of custom function models. We identify three types of function models: (i) functions defined in harness files that match project function signatures or serve as callbacks, (ii) replacement functions configured through Makefiles, and (iii) verified function contracts used to replace implementations. We exclude harness helper functions and models defined in shared stub files, whose attribution to individual harnesses is difficult to determine.

We statically analyze harness files and Makefiles to identify function models and the proportion of artifacts using them. For each model, we count its lines of code and precondition clauses. We report mean models per artifact, model sizes, and precondition counts across projects in each ecosystem.

3) Loop Bounds: Finally, we examine how practitioners configure loop bounds and contracts. For each artifact, we record whether it specifies global or loop-specific unwinding bounds, enables loop-contract checking, and its maximum configured unwinding bound. We also identify loops annotated with invariants using constructs such as CBMC's `__CPROVER_loop_invariant(expr)` and Kani's `#[kani::loop_invariant(expr)]`. We report the proportion of artifacts enabling loop-contract checking, the proportion of projects containing loop invariants, and the number of annotated loops per project.

D. RQ3: Verification Outcomes

Automated BMC systems must also produce verification artifacts that achieve meaningful coverage and reliable results at practical execution costs. RQ3 examines verification results, execution time, and code coverage to establish empirical baselines and identify limitations for verification automation.

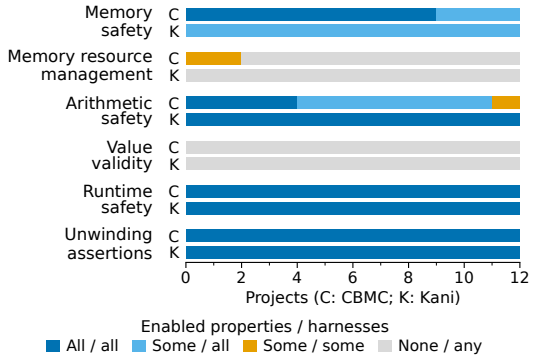


Fig. 2: Proportion of CBMC and Kani projects configuring built-in properties. The legend distinguishes projects enabling all or some properties across all or some harnesses.

1) Verification Results: We developed an automated pipeline to execute each artifact following its project's CI configuration. We run artifacts individually in Docker containers with a 15-minute timeout, consistent with the time limit adopted by the annual software verification competition [27]. We also impose a 6 GB memory limit based on available hardware. We did not increase these limits further because most artifacts completed well within them. We record whether verification succeeds, reports a property or unwinding assertion violation, or fails due to resource limits or other errors.

2) Verification Time: We measure verification time as the combined duration of symbolic execution and constraint solving, as reported by each tool. We exclude compilation, instrumentation, and reporting times because these stages differ between CBMC and Kani, limiting comparability.

3) Verification Coverage: We measure code coverage at both artifact and project levels. We rerun each artifact in coverage mode, using `cbmc-viewer` [29] to process CBMC results and Kani's coverage instrumentation for Rust projects [30]. For each artifact, we measure line coverage as the proportion of statically reachable lines covered under its input assumptions, function models, and loop configurations, as reported by the tool. At the project level, we measure function coverage as the proportion of project functions with at least one line covered by any artifact. Finally, we examine the relationship between verification time and the number of lines covered by each artifact.

V. Results

This section presents our findings and interpretations.

A. RQ1: Verification Properties

RQ1 characterizes built-in property adoption, diverse custom verification objectives, and specification errors.

1) Built-in Properties: Figure 2 shows substantial variation in the adoption of built-in property categories. Runtime safety checks (CBMC assertions and absence of Rust panics) and unwinding assertions are enabled in all projects. In contrast, no project enables CBMC's enum range checking or Kani's

invalid scalar value checking, while only two CBMC projects enable memory leak checking. These differences show that available built-in properties are not necessarily adopted.

We also find that default properties are more widely enabled than optional checks. Across CBMC and Kani projects, 96.9% and 100% of default properties are enabled, respectively, compared to 52.8% and 0% of non-default properties. No project explicitly disables a default property, although `FreeRTOS` lacks some newer default checks because it uses CBMC v5.95.1, predating their introduction in v6.0.0. These findings suggest that tool defaults and versions strongly influence verified properties, highlighting the importance of explicitly considering built-in checks in automated verification systems.

Finding 1.1: Default built-in properties are substantially more widely adopted than optional checks, suggesting that tool defaults strongly influence the properties selected for verification.

2) Custom Harness-Specific Properties: Table II presents our taxonomy of 12 custom property categories and their prevalence across projects. All Kani projects specify custom properties, compared to 83.3% of CBMC projects. Among adopting projects, custom properties appear in an average of 86.5% of Kani harnesses and 66.1% of CBMC harnesses. This widespread use highlights the importance of supporting project-specific specifications beyond built-in safety checks.

Our taxonomy identifies seven property categories in CBMC projects and eleven in Kani projects. The most prevalent categories concern function outputs: Result Numerical and Pointer Validity (P1), Result Correctness (P2), and Result Structural Validity (P3). Notably, Result Correctness appears in all Kani projects but only half of CBMC projects, with mean within-project harness shares of 69.2% and 35.2%, respectively, among adopting projects. Furthermore, only Kani projects in our dataset specify State Correctness, Expected Panic Behavior, Function Invocation, Supporting Lemma Correctness, and Model Correctness. Automated property generators must accommodate these diverse objectives.

Finding 1.2: Practitioners specify 12 categories of custom properties beyond built-in safety checks, with substantial differences in verification objectives across the two ecosystems.

3) Case Study on Custom Harness-Specific Properties: To illustrate these property categories, Listing 2 presents representative assertions and contracts from the studied projects, organized into four groups.

Properties of Result Variables (P1–P3): These properties constrain values returned by functions or written to output arguments. For example, a contract in `mldsa-native` requires polynomial coefficients updated by `poly_caddq` to remain within a specified range (P1). An `aws-c-common` harness verifies that saturating addition returns the exact sum when representable and the maximum integer otherwise (P2). Similarly, a `firecracker` harness checks structural invariants relating a token bucket's budget, capacity, and burst allowance (P3). These properties range from basic output validity to relationships that establish functional correctness.

Listing 2: Custom properties (P) and specification errors (E).

```
// Result variables
(P1) ensures(array_bound(a->coeffs, 0,
       MLDSA_N, 0, MLDSA_Q))
(P2) if (b > 0 && a > UINT64_MAX - b)
       assert(r == UINT64_MAX);
     else assert(r == a + b);
(P3) assert!(bucket.is_valid());

// State and input variables
(P4) assert(s2n_mem_get_page_size() > 0);
     assert(s2n_mem_get_page_size()
            <= UINT32_MAX);
(P5) assert_eq!(new_total,
       old_total + page_size as isize);
(P6) ensures(return_value != 0 ==>
       array_unchanged_or_zeroized_u8(
           c, MLKEM_INDCPA_BYTES))

// Behavioral and supporting properties
(P7) #[kani::should_panic]
     let _ = base.extend(field, packed);
// drop_count counts Droppable::drop() calls
(P8) assert_eq!(drop_count, 2);
(P9) __CPROVER_assert(
       pxNetworkBuffer != NULL, "...");
(P10) assert_eq!(bytes & align_mask, 0);

// Verification artifacts
(P11) __CPROVER_assert(
        PRV_UNLOCK_QUEUE_BOUND > 0, "...");
(P12) assert!(early <= late);

// Specification errors (names abbreviated)
(E1) ensures(forall(i, 0, len, r[i] ==
       (b == 0 ? x[i] : old(r)[i])))
(E2) assert((hi << 8) | lo == *dest);
(E9) let p = addr != A || addr != B; // A != B
     kani::assume(p == panic_mode);
```

Properties of State and Input Variables (P4–P6): These properties establish expectations on program state and the preservation of inputs. For example, an `s2n-tls` harness checks that the initialized memory page size falls within a valid range (P4), while a `spacewasm` harness verifies that allocating a page increases global memory accounting by exactly its size (P5). A contract in `mlkem-native` also requires the ciphertext buffer to remain unchanged or be entirely zeroized when encryption fails (P6). These properties are useful when correctness depends on state changes or preservation rather than function outputs alone.

Behavioral and Supporting Properties (P7–P10): These categories specify exceptional behavior, function interactions, and supporting correctness conditions. For example, a `zerocopy` harness verifies that extending an already dynamically sized layout causes a panic (P7), while a `spacewasm` harness checks that removing and dropping two vector elements invokes their destructors exactly twice (P8). A `FreeRTOS-Plus-TCP` harness verifies that ARP processing passes valid buffer pointers to a network-output callback (P9). Another `zerocopy` harness proves that multiplying an aligned element size preserves alignment when the product is representable (P10). These properties capture behavioral requirements and supporting facts beyond output assertions.

TABLE II: Prevalence of custom BMC property categories. Projects denotes the percentage of adopting projects; Mean H is the mean percentage of harnesses within those projects.

ID	Property	What Is Enforced	CBMC		Kani	
			Proj.	Mean H	Proj.	Mean H
	Presence of Custom Properties	*Project-specific requirements in harness assertions or enforced function postconditions.*	*83%*	*66%*	*100%*	*86%*
P1	Result Numerical or Pointer Validity	A result lies within numeric bounds, has an allowed value, or is a valid pointer.	83%	32%	83%	36%
P2	Result Correctness	A result satisfies a relationship with inputs or their pre-execution values.	50%	35%	100%	69%
P3	Result Structural Validity	Result-object fields satisfy relationships required for structural validity.	25%	60%	67%	17%
P4	State Numerical Range	A state variable lies within numeric bounds or has an allowed value.	17%	2%	0%	—
P5	State Correctness	State changes satisfy a specified relationship with function inputs.	0%	—	8%	6%
P6	Input and State Preservation	Inputs or state remain unchanged or are zeroized, or input objects retain validity.	42%	41%	42%	24%
P7	Expected Panic Behavior	An operation triggers an expected panic under specified conditions.	0%	—	33%	8%
P8	Function Invocation	Another function is invoked, possibly a specified number of times.	0%	—	8%	3%
P9	Caller Argument Correctness	Arguments passed to another function satisfy required constraints.	42%	26%	25%	2%
P10	Supporting Lemma Correctness	Mathematical facts or implications support an implementation or proof, independently of function results.	0%	—	17%	20%
P11	Harness and Setup Correctness	Checks proof configuration, type sizes or layouts, input construction, or harness-local values.	33%	4%	58%	10%
P12	Model Correctness	Checks function-model behavior or invariants independent of caller-supplied arguments.	0%	—	8%	3%

Properties of Verification Artifacts (P11–P12): These properties check the validity of verification setups and environment models. For example, a `FreeRTOS` harness checks that a configuration bound is positive before initializing queue-lock state (P11). Similarly, a `firecracker` harness verifies that its replacement model for `Instant::now` produces non-decreasing timestamps (P12). These examples highlight the need to validate verification artifacts themselves, rather than only the software they verify.

4) Property Specification Errors: Our review identified property-specification errors in 35 harnesses across ten projects. We distinguish two classes based on how they affect verification. The first involves incorrect property expressions that may alter or weaken the intended guarantees. These include (E1) aliased snapshots in preservation checks (8 harnesses), (E2) operator-precedence errors (5), (E3) assignments instead of equality checks (3), (E4) comparisons involving incorrect expressions (3), and (E5) incorrect enumeration-membership operators (2). Other cases involve (E6) checking an unchanged original object instead of its modified clones (1), (E7) an incorrect return-value range predicate (1), and (E8) a tautological Boolean return-value predicate (1).

The second class involves incorrect verification-domain constraints that can exclude relevant executions or make checks vacuous. These include (E9) incorrect proof-domain predicates that exclude ordinary correctness cases or fail to isolate expected-panic boundary cases (8), and (E10) contradictory or ineffective guards that leave selected assertions unreachable or vacuous (3). Unlike errors in property expressions, these mistakes can undermine verification even when the assertions themselves are correct. We provide each confirmed error and its source location in the supplementary materials, counting each affected harness once. These findings highlight the need for automated techniques that validate both property specifications and the conditions under which they are verified.

Listing 2 illustrates three errors. In `mlkem-native`, E1 snapshots a pointer rather than its contents, making the preservation branch a self-comparison. In `s2n-tls`, E2 parses equality before bitwise OR, so nonzero upper bytes can mask incorrect results. In `x86_64`, E9 is always true for distinct A, B, excluding all inputs from the non-panic proof. All 21 affected harnesses in four projects verify successfully in RQ3.

TABLE III: Verification artifact complexity.

Metric	CBMC	Kani
Harness and input models		
Mean harness size (LOC)	10.34	11.12
Projects with input preconditions	12/12 (100.0%)	10/12 (83.3%)
Harnesses with input preconditions	1,039 (78.7%)	224 (32.7%)
Mean preconditions / adopting harness	3.00	1.74
Function models		
Projects using function models	12/12 (100.0%)	3/12 (25.0%)
Harnesses using function models	591 (44.8%)	34 (5.0%)
Mean models / adopting harness	2.37	1.71
Contract replacements (% of models)	48.8%	74.1%
Loop bounds and contracts		
Projects with explicit numeric bounds	10/12 (83.3%)	10/12 (83.3%)
Harnesses with explicit numeric bounds	909 (68.9%)	173 (25.3%)
Mean largest bound / adopting harness	15.21	21.50
Projects with explicit loop invariants	2/12 (16.7%)	1/12 (8.3%)

Finding 1.3: Specification errors affect 35 harnesses across ten projects, demonstrating the need to validate both verification properties and their execution conditions.

B. RQ2: Artifact Complexity

Our RQ2 results show that practitioners construct small harnesses with few explicit preconditions, but frequently configure loop bounds and, in some projects, function models. These characteristics provide empirical baselines for automated harness construction. Table III summarizes our measurements.

1) Harness and Input Model Complexity: Most harness functions are small, with a median of seven lines of code (IQR: 4–14). Overall, 75% contain fewer than 15 lines, and only one exceeds 100 lines. These sizes are comparable to those reported for unit tests [31], [32], while systematically developed unit proofs average 20 lines [4]. Unlike unit testing, where multiple test cases are often written for each function, practitioners frequently use one harness per verification target.

Input preconditions are also common but generally few. Although 63% of artifacts (1,263) specify preconditions, the median is only one per harness, with counts ranging from 0 to 30. This contrasts with the five preconditions per unit proof reported by Amusuo et al. [4]. The remaining 742 artifacts (37%) specify no explicit input preconditions. This suggests automated harness generation can often begin with relatively simple input models, whose adequacy still requires validation.

Finding 2.1: Harnesses are typically small (median: seven lines) and use few explicit input preconditions (median: one), providing practical baselines for automated harness construction.

2) Function Models: Function modeling varies substantially between the two ecosystems. Only 5% of Kani artifacts (34/685) across three projects use custom function models, compared to 44.8% of CBMC artifacts (591/1,320) across all 12 projects. Among artifacts using models, the mean is 2.33 per artifact, comparable to the two models per unit proof reported by Amusuo et al. [4]. For artifacts containing written model bodies, the median total model-body size is nine lines (IQR: 3–14). These differences may reflect different verification targets or modeling strategies across ecosystems.

Function contracts are the most common form of identified models. Of the 1,456 models used across artifacts, 725 (49.8%) replace implementations with verified contracts, while 572 (39.3%) are defined directly in harness files. This demonstrates the importance of contract-based abstraction in existing BMC practices and motivates supporting both contract reuse and custom function modeling in automated verification systems.

Finding 2.2: Function models are considerably more prevalent in CBMC (44.8%) than Kani (5%) artifacts, while verified function contracts account for nearly half of all identified models.

3) Loop Bounds and Contracts: Explicit loop bounds are common, appearing in 54% of artifacts (1,082) across 20 projects. Of these, 62.7% specify only global bounds, while the others also or exclusively specify loop-specific limits. The maximum configured bound per artifact has a median of two but reaches 1,515. These findings highlight the importance of selecting appropriate execution bounds, since insufficient bounds can exclude executions containing property violations.

In contrast, loop contracts are rarely specified. Although 408 artifacts (20.3%) enable loop-contract checking, only three projects (`mlkem-native`, `mldsa-native`, and `hifitime`) contain loop invariant annotations, averaging 108.67 clauses per project. Their limited adoption may reflect the difficulty of specifying suitable invariants [33]. This contrast motivates automated techniques for inferring loop invariants and reducing reliance on manual unwinding bounds.

Finding 2.3: Explicit loop bounds appear in 54% of artifacts and are typically small (median maximum: two), while loop invariants are specified in only three projects.

Metric	CBMC	Kani
Completion		
Completed verification	1,305 (98.9%)	643 (93.9%)
within the main budget	1,281 (97.0%)	615 (89.8%)
only with the extended budget	24 (1.8%)	28 (4.1%)
Did not complete	15 (1.1%)	42 (6.1%)
Timeout (verification budget)	1 (6.7%)	24 (57.1%)
Out of memory	12 (80.0%)	17 (40.5%)
Build error	1 (6.7%)	0 (0.0%)
Required solver not installed	1 (6.7%)	0 (0.0%)
Unsupported language feature	0 (0.0%)	1 (2.4%)
Unsuccessful verdict		
Property violation	0 (0.0%)	0 (0.0%)
Only unwinding-assertion failures	2 (0.2%)	0 (0.0%)

TABLE IV: Verification outcomes per tool: We used a time and memory budget of 900 s and 6 GB per run.

C. RQ3: Verification Outcomes

Our RQ3 results show that most BMC artifacts complete verification quickly and cover most of their reachable code, but verification costs vary substantially and repository-level coverage decreases with project size.

1) Execution Outcomes: Table IV shows that 97.2% (1,948/2,005) of artifacts complete verification within the allocated resource limits. Of these, 1,946 complete without property violations, while two `FreeRTOS-Cellular-Interface` artifacts fail only unwinding assertions, indicating insufficient loop bounds. Of the remaining 57 artifacts, 54 exceed the time or memory limits and one fails to build.

Non-completion is about five times more frequent among Kani artifacts than CBMC artifacts (6.1% versus 1.1%), but failures are concentrated in a few projects. These include `xln` (12 timeouts among 28 artifacts), `zerocopy` (12 proofs disabled in CI for being too slow), and `hifitime` (5 memory-limit failures). Among the incomplete artifacts, 22 involve potentially expensive execution spaces, including unbounded loops or recursion and symbolically sized buffers. These findings motivate automated BMC techniques that reduce reliance on finite loop and recursion unwinding through inferred loop invariants, recursive function summaries, and sound abstractions for symbolically sized buffers.

2) Runtime and Resource Consumption: Most completed artifacts require little verification time. As shown in Figure 3, the median symbolic execution and constraint-solving time is 0.40 s for CBMC and 0.38 s for Kani. Overall, 64% complete these phases within one second, 87% within ten seconds, and 97% within one minute. Including overheads, the median wall-clock time is 0.62 s (CBMC: 0.56 s; Kani: 0.82 s).

Memory consumption differs more substantially between the tools. CBMC has a median peak memory consumption of 80 MB and a 90th percentile of 0.54 GB, compared to 376 MB and 2.5 GB for Kani. Kani's higher memory consumption may partly explain its higher non-completion rate, particularly for memory-limited artifacts. Executing the dataset, including

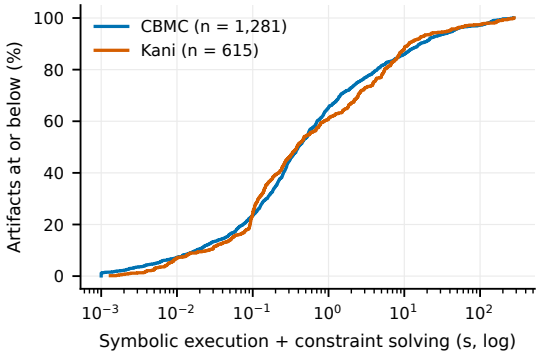


Fig. 3: Verification-time CDF for completed artifacts.

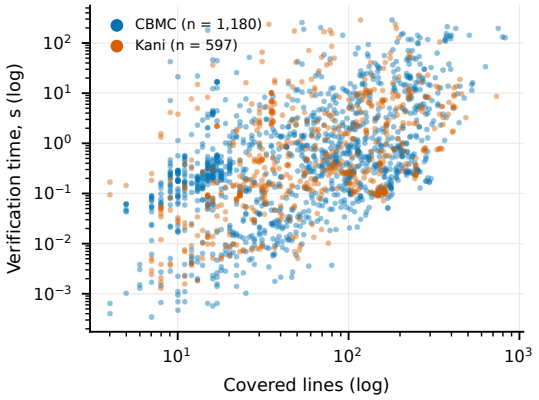


Fig. 4: Verification time against covered lines per completed artifact, shown on logarithmic axes.

compilation and coverage measurements, required 25.5 machine hours, with another 18.6 hours for the extended runs.

Finding 3.1: Most artifacts (97.2%) complete within the resource limits, with a median wall-clock verification time of 0.62 s. Resource failures are concentrated in a few projects, highlighting the need for targeted verification configurations.

We also examine whether verification time can be predicted from the number of lines covered by an artifact. Figure 4 shows a moderate positive correlation between verification time and covered lines (Spearman's $\rho = 0.50$). We fit power-law models for each tool using least squares in log space on 90% of the artifacts:

$$t_{\text{CBMC}} = 0.0035\, L^{1.22} \qquad t_{\text{Kani}} = 0.0056\, L^{1.09} \tag{2}$$

where t is verification time in seconds and L is the number of covered lines. The fitted exponents suggest slightly superlinear growth for both tools. However, artifacts covering similar amounts of code differ in verification time by up to four orders of magnitude. Covered lines explain only a limited share of runtime variation ($R^2 = 0.27$), with just 23% of held-out predictions falling within a factor of two. These findings suggest that factors such as loops, verification properties, and environment models may help predict verification costs.

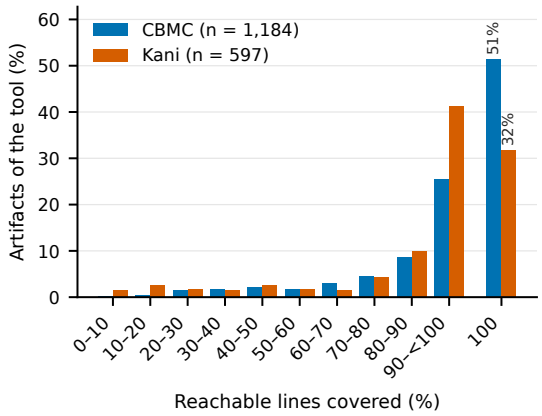


Fig. 5: Distribution of artifact-level line coverage for CBMC ($n = 1{,}184$) and Kani ($n = 597$).

Finding 3.2: Verification time increases slightly faster than linearly with covered code, but code size alone poorly predicts individual artifact runtime ($R^2 = 0.27$).

3) Artifact Coverage: Our results show that most verification artifacts cover a large proportion of their reachable code. Of the 1,896 artifacts completed within the initial budget, we obtained coverage measurements for 1,781. The remaining 115 lacked coverage results due to CBMC SMT backend errors (77), coverage timeouts (32), and memory exhaustion (6).

Figure 5 shows a median line coverage of 98% (CBMC: 100%; Kani: 96%). Overall, 45% of artifacts cover every reachable line, 24% cover less than 90%, and only 7% (124 artifacts) cover less than half. Full coverage is more common in CBMC than Kani (51% versus 32%), while low coverage is concentrated in `FreeRTOS-Plus-TCP` (49), `hifitime` (33), and `templar-contracts` (15).

We identified four explanations for the 124 artifacts with less than 50% coverage. (i) Narrow verification scenarios (63 artifacts, 51%) use fixed inputs or configurations that leave alternative paths unexplored. (ii) Coverage measurement artifacts (46, 37%) include unreachable functions in CBMC's coverage denominator. In 41 `FreeRTOS-Plus-TCP` artifacts, these are one-line placeholders for excluded functions; removing them raises coverage above 50% (median 96%). The remaining five retain functions whose addresses are taken. (iii) Contract-based abstraction (7, 6%) leaves callees of replaced function implementations unexecuted in `hifitime`. (iv) Vacuous verification (8, 6%) occurs when execution terminates before reaching the intended assertions.

4) Repository Coverage: Despite high artifact-level coverage, verification artifacts collectively cover a much smaller proportion of larger software projects. For example, artifacts in `FreeRTOS-Plus-TCP` statically reach 85% of its functions but execute at least one line in only 42%.

We find a strong negative correlation between project size and function coverage (Spearman's $\rho = -0.85$ across 23 projects; CBMC: -0.76; Kani: -0.71). Although projects contain between 35 and 12,570 functions, their verification

artifacts collectively cover only 1–264 functions per project. Consequently, smaller libraries often achieve near-complete function coverage, while larger projects cover a substantially smaller fraction of their code. We exclude `xln` because its artifacts cover no library functions.

These results reveal that thorough verification within individual artifact scopes does not necessarily translate into broad project-level coverage. This motivates automated techniques for identifying verification targets and expanding verification breadth, particularly in larger projects.

Finding 3.3: Artifacts achieve high coverage within their verification scope (median 98%), but repository-level function coverage decreases with project size. Low artifact coverage are caused by limited execution scope, coverage tool gaps, and harness errors.

VI. Discussion

We discuss implications for BMC use and automation.

A. Specifying Meaningful and Correct Properties

Our RQ1 results show that practitioners verify important software properties through built-in checks and custom specifications. Built-in checks are widely adopted, particularly when enabled by default, consistent with observations from prior work [34]. Custom properties also capture consequential requirements: `mlkem-native` verifies preservation or zeroization of failed cryptographic outputs, `firecracker` checks rate-limiter invariants, and `spacewasm` verifies correct memory deallocation. These examples show that meaningful requirements can be specified compactly and verified efficiently with BMC.

However, uneven adoption raises questions about whether practitioners verify all relevant properties for their projects. Categories such as State Correctness (P5), Supporting Lemma Correctness (P10), and Model Correctness (P12) appear in few projects, potentially reflecting domain-specific needs, specification difficulty, or limited practitioner awareness. Prior research on formal verification coverage also highlights that successful verification does not establish the completeness of specified properties [35]. Future studies could compare verified properties against requirements and historical defects to identify consequential gaps and develop techniques that recommend useful but overlooked properties.

Beyond selecting useful properties, practitioners must also specify them correctly. We identified specification errors in 35 harnesses across ten projects, yet 34 still completed verification successfully. These errors reinforce longstanding concerns about vacuous verification [36] and recent findings that LLM-generated specifications can pass verification without faithfully capturing intended requirements [37]. Prior research on vacuity detection and specification coverage [35], [36] provides a foundation for techniques that validate property semantics, input feasibility, and assertion reachability. Independent requirements, historical defects, and mutation testing could further help assess whether properties detect meaningful behavioral violations.

B. Scaling BMC Through Automation

Our RQ2 results show that practitioners construct small harnesses with few preconditions and relatively small loop bounds, suggesting that harness construction may be amenable to automation. However, it remains unclear whether these choices are sufficient for the intended properties or primarily reflect efforts to reduce verification cost. Prior studies have shown that incorrect assumptions, specifications, and function models can undermine verification guarantees [38]–[40]. How can automated systems make these choices without unnecessarily restricting the behaviors verified? AutoSOUP [41] provides an initial approach for deriving and validating verification choices for memory-safety proofs, but extending such techniques to the diverse functional properties observed in RQ1 remains an open challenge.

Automation could also address the limited repository-level coverage observed in RQ3, particularly in larger projects. However, target selection varies: `FreeRTOS-Kernel` concentrates 65% of its artifacts on one module, while `nonos-microkernel` distributes its artifacts across 42 of 52 modules. Whether these patterns reflect deliberate prioritization or limited verification effort remains unclear, motivating further studies of practitioners' target-selection decisions. Prior research has explored the feasibility of selecting software analysis targets using different vulnerability and testing signals [42]–[44], but their applicability to verification target selection remains unexplored. Alternatively, automated harness generation and compositional verification [45] could expand verification breadth, although correctly modeling interactions between components remains challenging. Future research could evaluate both approaches to determine how best to improve verification guarantees and coverage at practical cost.

C. Threats to Validity

External Validity: Our 24 selected CBMC and Kani projects may not represent practices across other BMC tools or software projects. Repository searches and filtering may have missed eligible projects, while differences between ecosystems may reflect programming languages or application domains rather than the tools themselves.

Construct Validity: We approximate artifact complexity using code size and construct counts and we exclude shared function models that are difficult to attribute to individual harnesses. Our LLM-assisted property classification and error identification may also be imperfect despite human review and significant inter-agent agreement.

Measurement and Interpretation: Reported high artifact-level coverage does not establish property adequacy or the correctness of verification guarantees. Also, observed associations between artifact characteristics and outcomes do not imply causation.

VII. Conclusion

We present an empirical study of BMC practices across 2,005 verification artifacts in 24 open-source projects. Our

findings show that practitioners verify diverse correctness and security properties using relatively small artifacts that often complete quickly, but specification errors and limited project-level coverage reveal important limitations. The property taxonomy, empirical measurements, and annotated dataset we contribute provide a foundation for designing AI-assisted verification systems that specify meaningful properties, construct appropriate verification models, and expand verification breadth. These advances can enable scalable and trustworthy software assurance as AI increasingly automates software development workflows.

VIII. Repository artifact.

All the scripts, data and results used to produce the results of this paper are published here: https://anonymous.4open.science/r/empirical-study-bounded-model-checking/

REFERENCES

[1] Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, J. Yu, and J. Chen, "Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 8, pp. 218:1–218:34, Oct. 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3716848

[2] M. A. 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*, ser. ASE '24. Association for Computing Machinery, pp. 1395–1407. [Online]. Available: https://dl.acm.org/doi/10.1145/3691620.3695512

[3] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal Methods in System Design*, vol. 19, no. 1, pp. 7–34, 2001. [Online]. Available: https://doi.org/10.1023/A:1011276507260

[4] P. C. Amusuo, O. Cochell, T. L. Lievre, P. V. Patil, A. Machiry, and J. C. Davis, "Do unit proofs work? an empirical study of compositional bounded model checking for memory safety verification," in *2026 IEEE/ACM 48th International Conference on Software Engineering*. [Online]. Available: http://arxiv.org/abs/2503.13762

[5] N. Chong, B. Cook, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle, "Code-level model checking in the software development workflow," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '20. Association for Computing Machinery, 2020, pp. 11–20. [Online]. Available: https://dl.acm.org/doi/10.1145/3377813.3381347

[6] T. Wu, S. Xiong, E. Manino, G. Stockwell, and L. C. Cordeiro. (2024) Verifying components of arm(r) confidential computing architecture with ESBMC. [Online]. Available: https://arxiv.org/abs/2406.04375v1

[7] D. Kroening and M. Tautschnig, "CBMC – c bounded model checker," in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds. Springer, 2014, pp. 389–391.

[8] kani. (2024) Getting started - the kani rust verifier. [Online]. Available: https://model-checking.github.io/kani/

[9] G. Takhar, B. Bijlani, P. Chatterjee, A. Lal, and S. Roy, "Memory-safety verification of open programs with angelic assumptions," *Proceedings of the ACM on Programming Languages*, vol. 9, pp. 312:1119–312:1147, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3763090

[10] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "ESBMC 5.0: an industrial-strength c model checker," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. Association for Computing Machinery, 2018, pp. 888–891. [Online]. Available: https://dl.acm.org/doi/10.1145/3238147.3240481

[11] L. Cordeiro, D. Kroening, and P. Schrammel, "JBMC: Bounded Model Checking for Java Bytecode," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer, M. Huisman, F. Kordon, and B. Steffen, Eds. Cham: Springer International Publishing, 2019, pp. 219–223.

[12] R. Delmas, Z. Hassan, Q. Hu, R. Kumar, F. R. Monteiro, T. Nguyen, A. Palacios, C. Val, M. Tautschnig, J. Adam, D. Schwartz-Narbonne, and C. Zech, "Kani: A Model Checker for Rust," in *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '26. New York, NY, USA: Association for Computing Machinery, Oct. 2026, pp. 4091–4103. [Online]. Available: https://dl.acm.org/doi/10.1145/3832783.3834499

[13] J.-C. Filliâtre, "Deductive software verification," vol. 13, no. 5, pp. 397–403. [Online]. Available: https://doi.org/10.1007/s10009-011-0211-0

[14] K. R. M. Leino, "Dafny: An Automatic Program Verifier for Functional Correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*, E. M. Clarke and A. Voronkov, Eds. Berlin, Heidelberg: Springer, 2010, pp. 348–370.

[15] M. Wenzel, L. C. Paulson, and T. Nipkow, "The isabelle framework," in *Theorem Proving in Higher Order Logics*, O. A. Mohamed, C. Muñoz, and S. Tahar, Eds. Springer, 2008, pp. 33–38.

[16] A. Brendel, A. Sivaraman, and T. Millstein, "Synthesizing Implication Lemmas for Interactive Theorem Proving," *Proceedings of the ACM on Programming Languages*, vol. 9, no. OOPSLA2, pp. 353:2254–353:2278, Oct. 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3763131

[17] J. S. Aitken, P. Gray, T. Melham, and M. Thomas, "Interactive Theorem Proving: An Empirical Study of User Activity," *Journal of Symbolic Computation*, vol. 25, no. 2, pp. 263–284, Feb. 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0747717197901759

[18] N. Chong, B. Cook, J. Eidelman, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle, "Code-level model checking in the software development workflow at amazon web services," *Software: Practice and Experience*, vol. 51, no. 4, pp. 772–797, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/spe.2949

[19] P. Schrammel, D. Kroening, M. Brain, R. Martins, T. Teige, and T. Bienmüller, "Incremental bounded model checking for embedded software," *Formal Aspects of Computing*, vol. 29, no. 5, pp. 911–931, 2017. [Online]. Available: https://doi.org/10.1007/s00165-017-0419-1

[20] D. Beyer, P.-C. Chien, B.-Y. Huang, N.-Z. Lee, and T. Lemberger, "HarnessForge: Automated Extraction of Verification Tasks from Industry-Scale Software Projects," in *Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, Jul. 2026, pp. 202–206. [Online]. Available: https://dl.acm.org/doi/10.1145/3803437.3806420

[21] M. Wang, Y. Ling, M. Gao, Y. Liu, and L. Huang, "KaPilot: LLM-assisted generation of Kani specifications for unsafe Rust verification," arXiv:2607.21957, 2026. [Online]. Available: https://arxiv.org/abs/2607.21957

[22] D. Matichuk, T. Murray, J. Andronick, R. Jeffery, G. Klein, and M. Staples, "Empirical study towards a leading indicator for cost of formal software verification," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 722–732, ISSN: 1558-1225. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7194620

[23] H. C. Estler, C. A. Furia, M. Nordio, M. Piccioni, and B. Meyer, "Contracts in Practice," in *FM 2014: Formal Methods*, C. Jones, P. Pihlajasaari, and J. Sun, Eds. Cham: Springer International Publishing, 2014, pp. 230–246.

[24] J. Dietrich, D. J. Pearce, K. Jezek, and P. Brada, "Contracts in the Wild: A Study of Java Programs," in *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), P. Müller, Ed., vol. 74. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, pp. 9:1–9:29. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2017.9

[25] Y. Demyanova, T. Pani, H. Veith, and F. Zuleger, "Empirical software metrics for benchmarking of verification tools," *Formal Methods in System Design*, vol. 50, no. 2, pp. 289–316, 2017. [Online]. Available: https://doi.org/10.1007/s10703-016-0264-5

[26] L. Bajczi, Z. Ádám, and Z. Micskei, "SV-COMP'25 Reproduction Report (Competition Contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Gurfinkel and M. Heule, Eds. Cham: Springer Nature Switzerland, 2025, pp. 187–191.

[27] D. Beyer and J. Strejček, "Evaluating Software Verifiers for C, Java, and SV-LIB," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Junges and G. Katz, Eds. Cham: Springer Nature Switzerland, 2026, pp. 461–502.

[28] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977, publisher: [Wiley, International Biometric Society]. [Online]. Available: https://www.jstor.org/stable/2529310

[29] "Reference manual - CBMC viewer," 2025. [Online]. Available: https://model-checking.github.io/cbmc-viewer/reference-manual/

[30] "Coverage - The Kani Rust Verifier." [Online]. Available: https://model-checking.github.io/kani/reference/experimental/coverage.html

[31] C. S. Yu, C. Treude, and M. Aniche, "Comprehending Test Code: An Empirical Study," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2019, pp. 501–512, iSSN: 2576-3148. [Online]. Available: https://ieeexplore.ieee.org/document/8918999

[32] R. Pan, T. Stennett, R. Pavuluri, N. Levin, A. Orso, and S. Sinha, "Hamster: A Large-Scale Study and Characterization of Developer-Written Tests," in *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '26. New York, NY, USA: Association for Computing Machinery, Jul. 2026, pp. 106–117. [Online]. Available: https://dl.acm.org/doi/10.1145/3786583.3786857

[33] C. A. Furia, B. Meyer, and S. Velder, "Loop invariants: Analysis, classification, and examples," *ACM Computing Surveys*, vol. 46, no. 3, pp. 1–51, Jan. 2014. [Online]. Available: http://dx.doi.org/10.1145/2506375

[34] G. Bennett, T. Hall, S. Counsell, E. Winter, and T. Shippey, "Do Developers Use Static Application Security Testing (SAST) Tools Straight Out of the Box? A large-scale Empirical Study," in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '24. New York, NY, USA: Association for Computing Machinery, Oct. 2024, pp. 454–460. [Online]. Available: https://dl.acm.org/doi/10.1145/3674805.3690750

[35] H. Chockler, O. Kupferman, and M. Y. Vardi, "Coverage Metrics for Formal Verification," in *Correct Hardware Design and Verification Methods*, D. Geist and E. Tronci, Eds. Berlin, Heidelberg: Springer, 2003, pp. 111–125.

[36] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient Detection of Vacuity in Temporal Model Checking," *Formal Methods in System Design*, vol. 18, no. 2, pp. 141–163, Mar. 2001. [Online]. Available: https://doi.org/10.1023/A:1008779610539

[37] W. Wang, S. Chen, Z. Sun, and S. Feng, "Are LLM-Generated Formal Specifications Faithful to Their Requirements? A Study on Bimodal Formal Specification Generation," Oct. 2026. [Online]. Available: https://arxiv.org/abs/2610.32053v1

[38] P. C. Amusuo, P. V. Patil, O. Cochell, T. Le Lievre, and J. C. Davis, "A unit proofing framework for code-level verification: A research agenda," in *2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2025, pp. 36–40. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/11023946

[39] S. Priya, X. Zhou, Y. Su, Y. Vizel, Y. Bao, and A. Gurfinkel, "Verifying verified code," *Innovations in Systems and Software Engineering*, vol. 18, no. 3, pp. 335–346, 2022. [Online]. Available: https://doi.org/10.1007/s11334-022-00443-9

[40] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy, "An empirical study on the correctness of formally verified distributed systems," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. Association for Computing Machinery, 2017, pp. 328–343. [Online]. Available: https://dl.acm.org/doi/10.1145/3064176.3064183

[41] P. C. Amusuo, R. Calvo, D. Anandayuvaraj, T. L. Lievre, K. Kolyakov, E. Jorgensen, A. Machiry, and J. C. Davis, "Autosoup: Safety-oriented unit proof generation for component-level memory-safety verification (extended report)," 2026. [Online]. Available: https://arxiv.org/abs/2605.10712

[42] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, "LEOPARD: Identifying vulnerable code for vulnerability assessment through program metrics," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 60–71, ISSN: 1558-1225. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8812029

[43] M. Zhang, J. Liu, F. Ma, H. Zhang, and Y. Jiang, "IntelliGen: Automatic driver synthesis for fuzz testing," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 318–327. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9402152

[44] F. Pecorelli, G. Grano, F. Palomba, H. C. Gall, and A. De Lucia, "Toward granular search-based automatic unit test case generation," *Empirical Software Engineering*, vol. 29, no. 4, p. 71, May 2024. [Online]. Available: https://doi.org/10.1007/s10664-024-10451-x

[45] B. Beckert, M. Kirsten, J. Klamroth, and M. Ulbrich, "Modular verification of JML contracts using bounded model checking," in *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*, T. Margaria and B. Steffen, Eds. Springer International Publishing, 2020, pp. 60–80.