
ESBMC: A SURVEY OF ITS EVOLUTION, INTEGRATION, AND FUTURE DIRECTIONS IN FORMAL SOFTWARE VERIFICATION

Pierre Dantas

Computer Science, The University of Manchester
Manchester, UK
pierre.dantas@manchester.ac.uk

Lucas Cordeiro

Computer Science, The University of Manchester
Manchester, UK
lucas.cordeiro@manchester.ac.uk

Waldir Junior

Electrical Engineering, Federal University of Amazonas (UFAM)
Manaus, AM, Brazil
waldirjr@ufam.edu.br

ABSTRACT

The Efficient SMT-based Context-Bounded Model Checker (ESBMC) has grown from a research prototype for verifying embedded ANSI-C software into one of the most versatile and industrially capable formal verification platforms available today. Since its first publication in 2009, ESBMC has undergone persistent evolution: expanding its verification techniques, widening its language support to nine front-ends, integrating industrial-strength Satisfiability Modulo Theories (SMT) solvers, and – most recently – coupling with Large Language Models (LLMs) and autonomous Artificial Intelligence (AI) agents. This survey traces the full trajectory of ESBMC from its original design principles to the state of the art in 2025–2026, documenting 43 awards at Competition on Software Verification (SV-COMP) and Competition on Software Testing (Test-Comp), its role as a formal verification backend for LLM-driven self-healing software and loop invariant generation, and the first industrial deployment of an integrated agentic model-checking architecture through the NVIDIA-OpenSMA framework, establishing ESBMC as a natively autonomous verification kernel rather than a passive validation backend. We synthesize its economic impact – over £9.3 million and €4.98 million in confirmed public research funding, the VeriBee spin-off, and a defense industrial deployment at Lockheed Martin – and conclude with a structured agenda of open challenges spanning scalability, neurosymbolic verification, counterexample intelligibility, cross-language verification, safety standards compliance, and open-source sustainability.

Keywords bounded model checking · SMT solving · formal verification · software model checking · ESBMC · LLM-assisted verification · k -induction · program analysis

1 Introduction

Software correctness is one of the oldest and most lasting challenges in computer science. As software systems increase in complexity and permeate safety-critical infrastructure – including medical devices [1], autonomous vehicles [2], blockchain smart contracts [3], and cloud-native microservices – the cost of defects escalates correspondingly. Formal verification, the discipline of mathematically proving that a program satisfies a specification, offers the strongest possible guarantees [4, 5]. Among its many techniques, Bounded Model Checking (BMC) [6, 7] has emerged as a practical and highly effective approach, balancing conceptual rigor with practical scalability.

The Efficient SMT-based Context-Bounded Model Checker (ESBMC) is one of the most significant tools to emerge from this tradition. First presented in 2009 as a framework for verifying embedded American National Standards Institute C (ANSI-C) programs using Satisfiability Modulo Theories (SMT) solvers [8], ESBMC has been continuously developed and extended since then [8, 9, 10, 11, 12] by a community anchored at the University of Manchester

(UK) and the Federal University of Amazonas (UFAM) (Brazil). Over that time, it has accumulated 43 awards at international software verification competitions, including 35 at Competition on Software Verification (SV-COMP) and 8 at Competition on Software Testing (Test-Comp) [13, 14, 15] (per-year tallies are verifiable in the annual competition organizers’ reports), support for nine programming languages [16, 3, 17, 18], and a pioneering integration with Large Language Models (LLMs) for automated bug repair [19].

This survey makes the following contributions:

1. Provide a sequential account of ESBMC’s origins, design decisions, and major milestones from 2009 through version 7.7 in 2025, connecting early theoretical choices to existing capabilities.
2. Analyze ESBMC’s core verification techniques, with emphasis on the evolution and integration of BMC, k -induction, incremental solving, floating-point arithmetic, and concurrency handling as an SMT-based, multi-language platform.
3. Examine the combination of Artificial Intelligence (AI)/LLM technologies with ESBMC, focusing on its function as a formal verification kernel in autonomous software engineering pipelines – including automated vulnerability repair [20], LLM-generated loop invariants [21], and the SpecVerify cyber-physical deployment [22] – and document the first industrial deployment of an integrated agentic model-checking architecture predating parallel external formulations [23, 24].
4. Synthesize evidence of technology transfer and industrial deployment, including confirmed bug findings in the Ethereum Consensus Specification (ECS), Decentralized Finance (DeFi) smart contracts, and adoption in defense industry contexts.
5. Quantify the economic and social impact of ESBMC, drawing on research funding, commercial spin-offs, and the cost scope of software defects and formal methods.
6. Identify open challenges and future research directions, with an agenda spanning scalability, neurosymbolic verification, counterexample intelligibility, cross-language verification, real-time and timing analysis, standards compliance, and sustainability under commercialization.

Author positionality This survey is written in part by ESBMC’s founding and lead developer (L.C.) and by collaborating researchers who have collectively co-authored the majority of the primary ESBMC papers surveyed in Sections 4–8. It therefore combines the depth of insider knowledge with the limitations of perspective inherent in insider authorship. We have sought to support all substantive claims with independently verifiable sources and to apply the same critical standards to ESBMC’s results as we do to competing tools; where primary sources authored by the ESBMC team are the only available evidence – as is structurally unavoidable for a single-tool survey – this is documented in Section 1.1. Section 1.1 explicitly discloses the self-citation rate, and the Acknowledgments disclose the conflict of interest arising from Cordeiro’s roles as a tool developer and VeriBee co-founder. Readers are encouraged to consult the independently validated SV-COMP results [14] and the primary technical papers cited throughout for independent verification.

The remainder of this survey is structured as follows: Section 1.1 describes the survey methodology, literature search strategy, and taxonomy of the BMC tool landscape; Section 2 introduces the conceptual foundations of BMC and SMT solving; Section 3 describes the origins and founding motivations of ESBMC; Section 4 traces its chronological evolution across sixteen years of development; Section 5 analyses the core verification techniques and SMT integration; Section 6 covers the expansion to new programming languages; Section 7 examines competition performance and industrial adoption; Section 8 surveys the combination with AI, LLMs, and autonomous agents; Section 9 quantifies the economic and social impact; Section 10 documents spin-offs, technology transfer, and important case studies; Section 11 identifies open challenges and future research directions; and Section 12 concludes.

1.1 Paper Selection Statistics

We queried all five databases on 21 May 2026 – three directly via public Application Programming Interfaces (APIs) (arXiv, Digital Bibliography & Library Project (DBLP), OpenAlex) and two (Institute of Electrical and Electronics Engineers (IEEE) Xplore, Association for Computing Machinery (ACM) Digital Library) via the OpenAlex publisher-stratified breakdown, which provides broad coverage of their indexed content without requiring institutional login. A full audit log and per-query counts are available at https://github.com/ssvlab/ssvlab.github.io/tree/master/papers_audit/2026-05-ESBMC-Survey/search_audit.

- **Open-access automated results:** arXiv returned 64 records on the primary string and 708 on the supplementary string (772 total). DBLP returned approximately 588 records on primary terms and 31 on supplementary terms (approximately 619 total, estimated after de-aliasing constituent term queries).

- **Publisher-stratified results via OpenAlex:** SpringerLink contributed approximately 630 primary and 20 supplementary records (650 total). IEEE Xplore contributed approximately 30 primary and 5 supplementary records (35 total). ACM Digital Library contributed approximately 55 primary and 6 supplementary records (61 total). The combined pre-deduplication total across all five sources is approximately 2136 candidate records.
- **Deduplication and screening:** We performed manual deduplication based on title, author, and publication-venue matching across the five sources. After removing cross-library duplicates – we measured approximately 26 % overlap, consistent with rates reported for multi-database software-engineering reviews [25] – approximately 1602 unique records remained. Papers were then screened against the inclusion and exclusion criteria in Section 1.2. The principal reasons for exclusion were hardware-only model checking (E1), pure propositional Boolean Satisfiability (SAT) content (E2), and superseded or duplicate versions (E3). Papers excluded under criterion E4 (unrefereed grey literature not traceable to a named funder or recognized venue) were the next-largest category; we retained grey literature traceable to named funders (UKRI, EU CORDIS), recognized standards bodies, and competition proceedings (SV-COMP, Test-Comp) via forward and backward citation chaining from retained primary papers.
- **Final corpus:** We compiled a total corpus of 107 sources, all of which we cite in this survey; we verified this figure by enumerating distinct cite keys appearing in the body text. We added items published between April 2025 and May 2026 – including SV-COMP/Test-Comp 2025 results, the agentic model checking preprint [23], the NVIDIA-OpenSMA version-control record [24], and arXiv preprints cited in Sections 8 to 11 – by directed update, and these items are not part of the systematic protocol window. Figure 1 summarises the stages of the screening funnel.

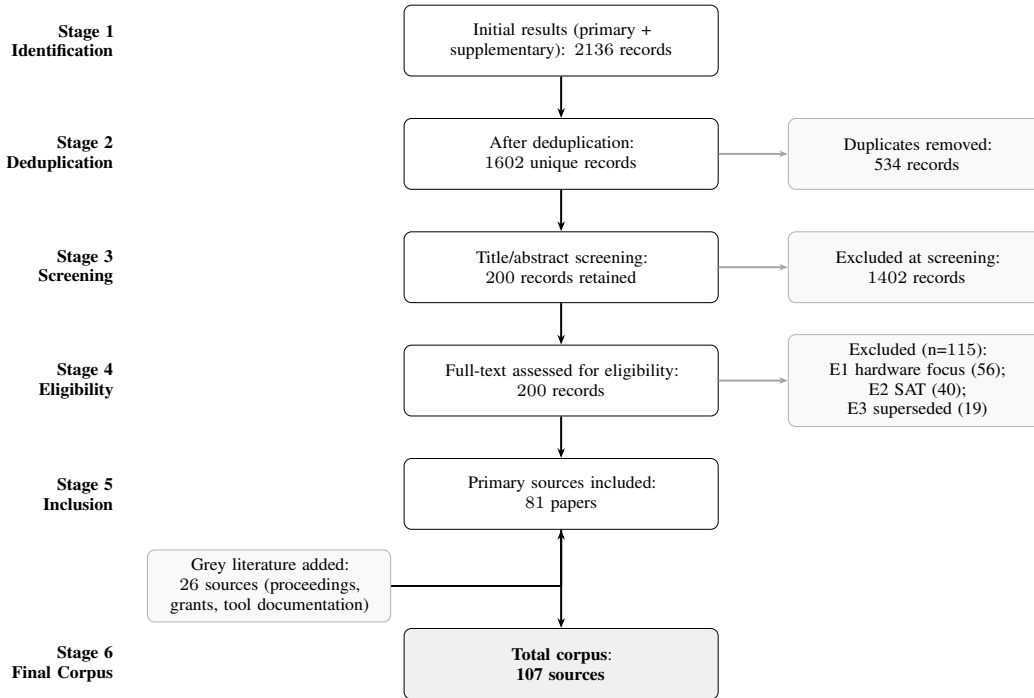


Figure 1: PRISMA-style search and selection flow. Combined primary and supplementary searches across arXiv, DBLP, IEEE Xplore, ACM Digital Library, and SpringerLink returned 2136 candidate records; 1602 remained after cross-library deduplication; 200 advanced to full-text review; 81 were accepted as primary sources. We added a further 26 grey literature sources, bringing the final corpus to 107 sources.

Of the 107 sources in the final corpus, 26 (24 %) are co-authored by members of the ESBMC research team or are institutional pages maintained by the authoring group (satisfying criterion I1). We structurally expect this percentage for a single-tool survey: technical descriptions of ESBMC’s own capabilities have no alternative primary source. It is disclosed here in the interest of transparency. All performance claims that admit independent corroboration – SV-COMP rankings [14], grant amounts traceable to the UK Research and Innovation (UKRI) Gateway to Research or EU Community Research and Development Information Service (CORDIS) records – are additionally supported by sources not co-authored by the ESBMC team.

1.2 Survey Scope and Type

This survey is a structured narrative of a single tool ecosystem, not a systematic mapping study of the BMC field as a whole. Its primary objective is depth over breadth: an expert-annotated account of ESBMC’s technical evolution, industrial deployment, and future research agenda. The systematic search component ensures that we do not inadvertently omit contextually important competing tools and theoretical antecedents; it does not aim to produce a comprehensive, count-based characterization of the literature. We do not attempt quantitative meta-analysis because the evidence base is not sufficiently homogeneous across evaluation protocols, property categories, or tool versions to support it.

1.3 BMC Tool Landscape: Taxonomy

Table 1 positions ESBMC within the landscape of the most active BMC and related software verifiers, evaluated against dimensions that directly reflect ESBMC’s principal design choices. Coverage is limited to tools that have participated in SV-COMP [13, 14] since 2012 or are otherwise widely cited in the software BMC literature; we exclude tools operating exclusively on hardware description languages and assess capabilities as of 2025.

Table 1: BMC and related software verification tools: taxonomy as of 2025. **Tech.:** BMC = Bounded Model Checking; k -ind = k -induction; CEGAR = Counterexample-Guided Abstraction Refinement; Pred = predicate abstraction; Auto = automata-theoretic; CHC = Constrained Horn Clause; Sym = symbolic execution; ES = explicit-state. **Langs:** distinct verified source languages (IR variants excluded). **Concur.:** concurrency support. **LLM:** confirmed LLM integration in published work. **SVC:** active SV-COMP participation in 2024 or 2025. yes (✓); partial (≈); not supported (–)

Tool	Technology	SMT Backends	Langs	Concur.	k -Ind.	LLM	SVC
ESBMC [8, 9]	BMC + k -ind	Z3, Bitwuzla, MathSAT, CVC5, Yices	9 [†]	✓	✓	✓	✓
CBMC [26]	BMC	SAT (CaDiCaL); Z3*	4	✓	–	–	✓
CPAchecker [27]	CEGAR + Pred + BMC + k -ind	MathSAT, Z3, SMTInterpol	1	✓	✓	–	✓
Ultimate Auto. [28]	Auto + CEGAR	Z3, MathSAT	1	≈	–	–	✓
2LS [29]	k -ind + Abstr. Interp.	SAT (CProver)	1	≈	✓	–	≈
Symbiotic [30]	Sym + BMC + Slicing	Z3	1	–	–	–	✓
DIVINE [31]	ES	– (native)	2	✓	–	–	✓
Theta [32]	CEGAR + BMC	Z3, MathSAT	1	–	–	–	✓
Kani [33]	BMC	SAT (via CBMC)	1	≈	–	–	–
SeaHorn [34]	CHC	Z3	2	–	–	–	≈

* Z3 used for certain property categories only (SMT backend not primary competitive configuration).

[†] C++03 and C++11+ are counted as distinct front-ends; treating them as one language family gives 8 distinct languages.

Four characteristics collectively distinguish ESBMC from its closest competitors. First, its **native multi-solver SMT backend** dispatches to six interchangeable solvers covering bit-vectors (Bitwuzla, Boolector), floating-point arithmetic (Bitwuzla, MathSAT), general arithmetic (Z3), and strings and algebraic datatypes (CVC5) [35, 36, 37, 38, 39]; no other tool in Table 1 offers comparable solver breadth with automatic theory-guided selection. Second, its **nine-language portfolio** is the widest in the field, spanning embedded C/C++, graphical processing unit (GPU) programs (Compute Unified Device Architecture (CUDA)), smart contracts (Solidity), Java Virtual Machine (JVM) languages (Kotlin via Jimple), capability hardware (Capability Hardware Enhanced RISC Instructions (CHERI) C), and systems languages (Python, Rust). Third, to the best of the authors’ knowledge, it is currently the **only open-source BMC tool with a published and evaluated LLM integration** for automated vulnerability repair [19], loop invariant generation [21], and specification translation [22]. Fourth, the combination of **BMC and k -induction in a single engine** – shared with CPAchecker and 2LS but absent from Bounded Model Checking for ANSI-C Programs (CBMC), Symbiotic, and SeaHorn – enables both counterexample finding and unbounded correctness proofs without requiring tool switching.

2 Background: BMC and SMT Solving

Model checking is an automated technique for verifying finite-state reactive systems by exhaustively exploring their reachable state spaces [40, 41, 5]. Classical symbolic model checking using Binary Decision Diagrams (BDDs) [42] suffered from state-space explosion for large systems [43]. BMC, introduced in 1999 [6], addressed this by restricting verification to traces of up to a fixed depth k [7]: we unroll a program k times, negate a safety property, and hand the

resulting formula to a SAT or SMT solver, where a satisfying assignment constitutes a counterexample. BMC’s key pragmatic advantage is its ability to find bugs quickly when they exist within a shallow execution depth. Complementary techniques such as k -induction address the principal limitation of this approach – the inability to prove correctness for unbounded programs [44, 45], Property Directed Reachability (PDR), and Incremental Construction of Inductive Clauses for Indubitable Correctness (IC3).

SMT extends propositional SAT with decision procedures for rich background theories: linear arithmetic, bit-vectors, arrays, floating-point arithmetic, and strings [46, 5, 47]. Solvers such as Z3 [35], Bitwuzla [36], CVC5 [38], MathSAT [37], Yices [39], and Boolector [48] have become the workhorses of modern program verification. ESBMC’s design from the outset was SMT-centric [8], separating it from SAT-based predecessors [49] and permitting more precise and expressive reasoning about program semantics. ESBMC shares a common origin with CBMC [26], originally developed at Carnegie Mellon University, and the developers initially built both tools on the CProver software infrastructure [8]. However, ESBMC progressively replaced and extended virtually every component of this shared core [8, 9], driven by the goal of adopting SMT reasoning natively rather than as a post-processing step on SAT encodings.

3 Origins and Founding Motivations (2008–2009)

Around 2008, a recognized gap emerged in the verification landscape: the lack of a purpose-built, SMT-native BMC for embedded C software. Existing tools either relied exclusively on SAT solvers (e.g. CBMC [26, 49]), were designed for hardware verification [50], or lacked support for the numeric and pointer-rich programs prevalent in embedded systems.

At the time, SMT solvers had matured significantly [35], supporting theories such as linear arithmetic over integers and reals, bit-vectors, and arrays [46, 5]. The insight behind ESBMC was to exploit this expressiveness directly. Rather than bit-blasting all program variables and arithmetic into propositional logic, the verifier would encode them symbolically using the appropriate SMT theory, yielding smaller and more efficiently solvable formulae [8].

The tool was formally introduced in the paper “SMT-based BMC for Embedded ANSI-C Software” [51], presented at Automated Software Engineering (ASE) 2009, which set out the core architectural decisions:

- **SMT-native encoding:** Program state is encoded directly into SMT formulae [46, 5], exploiting theories for integer arithmetic, bit-vectors, and arrays rather than collapsing everything to propositional logic.
- **Embedded C focus:** The initial tool targeted ANSI-C with features common in embedded software: fixed-point arithmetic, pointer manipulation, bitfield operations, and structs.
- **CProver front-end:** ESBMC adopted the CBMC/CProver front-end [26] for parsing and Intermediate Representation (IR) generation, but replaced the back-end entirely with its own SMT translation layer.
- **Multiple SMT solver support:** From the outset, ESBMC was designed to support multiple SMT solvers interchangeably [35, 37, 39, 36], recognizing that no single solver dominates across all problem categories.

A companion paper appeared in the IEEE Transactions on Software Engineering [8], providing a more extensive treatment of the conceptual framework and its empirical study.

Lucas Cordeiro, a doctoral researcher under Bernd Fischer at the University of Southampton (UK), conducted the first development, with Joao Marques-Silva at University College Dublin providing expertise in SAT/SMT solving [46]. Subsequently, Cordeiro established the Systems and Software Verification LAB (SSVLab) [15] at the UFAM (Brazil), before joining the University of Manchester, where he became Director of the ARM Centre of Excellence in 2021.

4 Chronological Evolution of ESBMC

Figure 2 presents ESBMC’s sixteen-year development path. The pattern visible is one of foundational theoretical contributions in the early years (SMT-native encoding, k -induction, floating-point theory, concurrency), followed by progressive industrialization and competitive hardening, and culminating in a rapid burst of language extensions and AI integration from 2022 onwards. This trajectory has remained shaped in large part by continuous participation in the International SV-COMP [13, 14] – a public annual benchmark that offers rigorous external feedback on correctness and performance – against which ESBMC has accumulated 43 awards (35 at SV-COMP and 8 at Test-Comp) since its debut in 2012 [13, 14, 15].

4.1 Early Versions and SV-COMP Debut (2010–2014)

Following the 2009 publication, ESBMC entered a rapid development phase. By 2012, the authors submitted ESBMC 1.17 to the inaugural SV-COMP 2012 at Tools and Algorithms for the Construction and Analysis of Systems

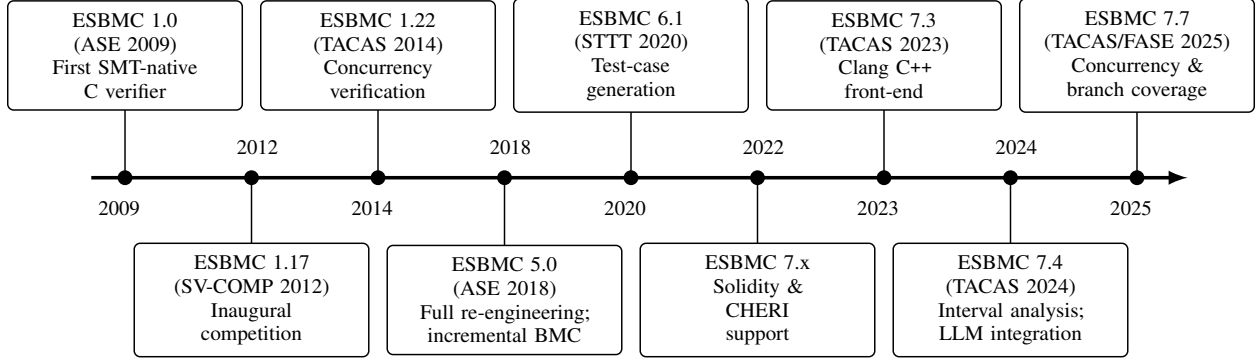


Figure 2: Chronological milestones of ESBMC major releases from its first publication at ASE 2009 through version 7.7 in 2025. Labels alternate above and below the axis to prevent overlap. Each marker corresponds to a peer-reviewed venue or substantive feature release [8, 13, 52, 9, 53, 10, 11, 12, 54]

(TACAS) [13], marking the tool’s first head-to-head evaluation against other state-of-the-art verifiers and establishing a pattern of competitive engagement that would drive quality improvements for over a decade. ESBMC 1.22 was subsequently submitted as a competition contribution to TACAS 2014 [52].

A major theoretical advance in this period was the extension to multi-threaded software verification, formally presented at International Conference on Software Engineering (ICSE) 2011 [55]. Rather than enumerating all possible thread interleavings – exponential in the number of threads and context switches – ESBMC adopted context-BMC: each verification query bounds the number of context switches per execution trace, making the search space finite and tractable. This strategy facilitates systematic detection of data races, deadlocks, and atomicity violations, and represented one of the first SMT-based treatments of concurrent ANSI-C verification.

Equally significant was the integration of k -induction [44, 45] as a complement to BMC: while BMC finds counterexamples within a bound k , k -induction enables proof of correctness for all executions, not just up to k . This approach proved a defining feature of ESBMC, separating it from other methods in verification competitions. Timing-constraint verification was also achieved through encoding timing requirements as integer constraints [56], eliminating the need for special real-time extensions.

4.2 Maturation and Industrial Strengthening (2015–2018)

ESBMC 5.0, which the authors presented at ASE 2018 [9], thoroughly re-engineered almost all components: it largely superseded the CProver inheritance; provided comprehensive C property checks (array bounds, divisions by zero, pointer safety, arithmetic overflow, memory leaks, deadlocks, data races); and introduced incremental BMC, extending the unrolling depth but without re-solving unchanged formula fragments – greatly improving performance on deep benchmarks.

An important capability consolidated in this period was floating-point arithmetic verification. ESBMC’s encoding of IEEE 754 floating-point numbers via the SMT floating-point theory was systematically evaluated [57], demonstrating that native SMT floating-point encoding outperforms bit-blasting alternatives on embedded numerical assessments drawn from SV-COMP. This capability is notably important for safety-critical software in avionics, automotive control, and industrial automation, where floating-point rounding errors are a common and difficult-to-test source of specification violations.

4.3 Automated Test Generation and Software Tools for Technology Transfer (STTT) Publication (2020)

ESBMC 6.1 [53], published in the International Journal on STTT, extended the tool to automated test case generation: BMC-derived counterexamples are systematically converted into executable test cases, bridging formal verification and software testing practice. This extension (the automated test case generation capability) also enabled participation in Test-Comp alongside SV-COMP. The generated test cases target branch- and condition-coverage criteria, complementing the safety-property orientation of BMC with a structural-coverage perspective more familiar to practitioners who rely on testing rather than formal proof. In Test-Comp, the benchmark evaluates tools on their ability to produce test suites that maximize test coverage on a standardized test set; ESBMC’s entry demonstrated that a model-checking back-end can compete directly with dedicated test-generation tools, without sacrificing the counterexample-quality guarantees that SMT-based reasoning provides.

4.4 C++ Model Checking and Platform Extensions (2021–2022)

The correctness of ESBMC’s C++ verification features was comprehensively formalized in the journal “Software Testing, Verification and Reliability” [16], covering the full object-oriented feature set found in industrial codebases: templates, inheritance, virtual dispatch, dynamic memory management, and exception handling. This work presented ESBMC as a capable verifier not only for C but for modern C++ programs – a prerequisite for its subsequent competition entries in C++ categories at SV-COMP.

This period likewise saw ESBMC extended to two novel and contrasting application domains. ESBMC-CHERI [58], presented at International Symposium on Software Testing and Analysis (ISSTA) 2022, adapted ESBMC for the CHERI capability-hardware architecture, in which every pointer carries embedded bounds and permission tags enforced at the hardware level. The extension translates CHERI’s capability semantics into SMT constraints, enabling verification that C programs respect capability provenance and never exceed pointer bounds – a critical correctness property for compartmentalized, memory-safe systems that the Defense Advanced Research Projects Agency (DARPA) and UKRI funded hardware security programs are exploring.

A parallel extension, ESBMC-Solidity [3], presented at ICSE 2022, targeted Ethereum smart contracts written in Solidity. Smart contracts manage major financial value yet are immutable once deployed, making pre-deployment formal verification especially valuable. ESBMC-Solidity translates Solidity source into an ANSI-C model and applies SMT-based BMC to detect reentrancy attacks, integer overflows, and violated contract assertions – classes of vulnerability responsible for hundreds of millions of dollars of losses in deployed contracts.

4.5 C++ Modernisation with Clang Abstract Syntax Tree (AST) (2023)

ESBMC v7.3, at TACAS 2023 [10], replaced the legacy C++ front-end with one built directly on the Clang compiler’s AST, enabling correct parsing of all modern C++ dialects up to C++20, including lambda expressions, range-based for loops, and structured bindings. The legacy CProver-derived front-end had handled a C++03 subset adequately but struggled with post-2011 features, forcing users to rewrite or simplify source code before verification.

By delegating parsing and semantic analysis to Clang – one of the most complete and actively maintained C++ compilers available – ESBMC gained access to the full C++11/14/17/20 feature set without duplicating compiler engineering effort. The practical consequence is that industrial codebases written in modern C++ can now be verified without syntactic workarounds, substantially broadening the tool’s applicability to embedded and systems software.

4.6 Interval Analysis and Performance Optimizations (2024)

ESBMC v7.4, at TACAS 2024 [11], introduced a static interval analysis pass that prunes infeasible execution paths and simplifies SMT formulae before solver invocation, yielding up to 7 % more provably correct programs under k -induction. Interval analysis computes over-approximations of variable ranges at each program point, injecting these ranges as additional assumptions that narrow the search space the SMT solver must explore.

This interval analysis pass proves especially effective for k -induction, where tighter invariants directly strengthen the induction hypothesis and enable users to prove more properties without increasing the unrolling bound. At SV-COMP 2024, ESBMC v7.4 achieved 4th place overall among 30 participating verifiers [14, 11] – a commercially significant profile for inclusion into Continuous Integration and Continuous Deployment (CI/CD) pipelines where wall-clock verification time is a hard constraint.

4.7 Concurrent Verification and Branch Coverage (2025)

At TACAS 2025 [12] and Fundamental Approaches to Software Engineering (FASE) 2025 [54], ESBMC v7.7 delivered: (1) a new thread scheduling algorithm combined with incremental SMT solving and enhanced partial order reduction for multi-threaded verification; and (2) Control Flow Graph (CFG)-based branch coverage instrumentation, directly targeting structural coverage requirements of standards such as Software Considerations in Airborne Systems and Equipment Certification (DO-178C) (avionics) and International Electrotechnical Commission (IEC) 61508 (functional safety). The scheduling algorithm models thread states as symbolic transition systems, enabling SMT-based independence checking at each scheduling decision and dramatically reducing the number of interleavings the tool must explicitly explore.

The incremental solving strategy preserves the solver state across successive context-switch explorations, avoiding redundant re-solving of the shared-memory access structure common to many interleavings. Together, these advances enabled ESBMC to compete in the concurrency and branch-coverage categories of SV-COMP 2025 and Test-Comp 2025, extending the tool’s reach into the structural testing domain that safety certification auditors demand.

5 Evolution of Verification Techniques and SMT Integration

5.1 SMT Solver Portfolio

A defining architectural decision of ESBMC is its solver-agnostic design. The tool maintains a uniform internal representation of SMT formulae and dispatches to the most appropriate solver for the problem at hand. As of 2025, we listed the supported solvers in Table 2. This portfolio allows ESBMC to select the best-performing solver for a given property category, a feature heavily leveraged in competition submissions. Incremental API support – relevant to the `-incremental-bmc` mode discussed in Section 5.3 – varies by solver: Z3, Bitwuzla, MathSAT, CVC5, and Yices 2 provide full push/pop stack semantics, while Boolector offers only a partial scope-reset mechanism.

Table 2: ESBMC supported SMT solvers and their key strengths

Solver	Key Strengths
Z3 [35]	General-purpose; excellent for quantifier-free arithmetic and arrays
Bitwuzla [36]	Bit-vectors and floating-point; successor to Boolector
Boolector [48]	Bit-vector arithmetic; historically strong on hardware-style benchmarks
MathSAT [37]	Interpolation; model generation; quantified arithmetic
CVC5 [38]	Strings; algebraic datatypes; advanced arithmetic theories
Yices 2 [39]	Speed on linear arithmetic and bit-vectors

Modern SMT solvers use the Davis-Putnam-Logemann-Loveland (DPLL)(T) paradigm [46], combining a SAT core and theory solvers for arithmetic, bit-vectors, and arrays. ESBMC does not implement DPLL(T) itself; instead, it interacts with these solvers by dispatching a uniform SMT formula to a portfolio of solvers. At the dispatch boundary, a thin translation plugin serializes the formula to a solver-specific API or to Satisfiability Modulo Theories Library (SMT-LIB) [5, 47], which allows new solvers to integrate without modifying the verification engine (Figure 3).

Solver choice impacts competition performance. Bitwuzla [36] and Boolector [48] are strong on bit-precise properties, Z3 [35] for arithmetic, MathSAT [37] for interpolation, and CVC5 [38] for strings and datatypes. No single solver dominates all SV-COMP categories [14], so ESBMC’s flexible solver switching is a key competitive advantage [14, 15].

ESBMC balances automated and manual solver selection. By default, it heuristically picks the best solver for the program and property, such as Bitwuzla or Boolector for bit-vectors, Z3 or MathSAT for arithmetic, and CVC5 for strings. The developers inform these choices using performance data from benchmarking and SV-COMP. Users can explicitly select solvers via command-line options or configuration files, overriding defaults for experiments or manufacturing needs. In competitions, custom scripts can optimize solver choice. This flexibility permits both automated and manual selection of solvers for advanced workflows.

5.2 k -Induction

The k -induction proof rule [44, 45] extends BMC from a bug-finding technique into a proof method. The algorithm advances in two phases: (i) the base case unrolls the program for k steps and checks that the negation of the property is unsatisfiable – ruling out counterexamples of length up to k , and (ii) the inductive step assumes the property holds for k consecutive states (not necessarily reachable ones) and attempts to prove it holds for the $(k + 1)$ -th; if this SMT query is also unsatisfiable, we prove the property for all execution lengths. The two phases together constitute a sound and relatively complete verification procedure for programs with finite loop bounds.

A major challenge is that many real-world properties are not k -inductive as stated: the inductive step may fail not because the property is false, but because the induction hypothesis is too weak – it admits unreachable states from which a violation seems possible. The standard remedy is to strengthen the hypothesis with auxiliary invariants.

ESBMC draws on two complementary sources to strengthen the induction hypothesis when the inductive step fails, as illustrated in Figure 4. First, a static interval analysis pass [11] computes over-approximations of variable ranges at each program point and injects these as additional assumptions before solver invocation, pruning spurious abstract states and making more properties inductive – a lightweight, fully automated procedure. Second, LLM-generated loop invariants [21] – and corroborating results from [60], which evaluates LLM-based loop invariant generation independently of ESBMC – can be supplied as candidate auxiliary assertions: ESBMC checks their validity with SMT and, if they pass, uses them to strengthen the induction hypothesis. This integration of neural candidate generation with formal validity checking (explored further in Section 8) illustrates the wider trend of merging statistical and deductive reasoning in program verification [22].

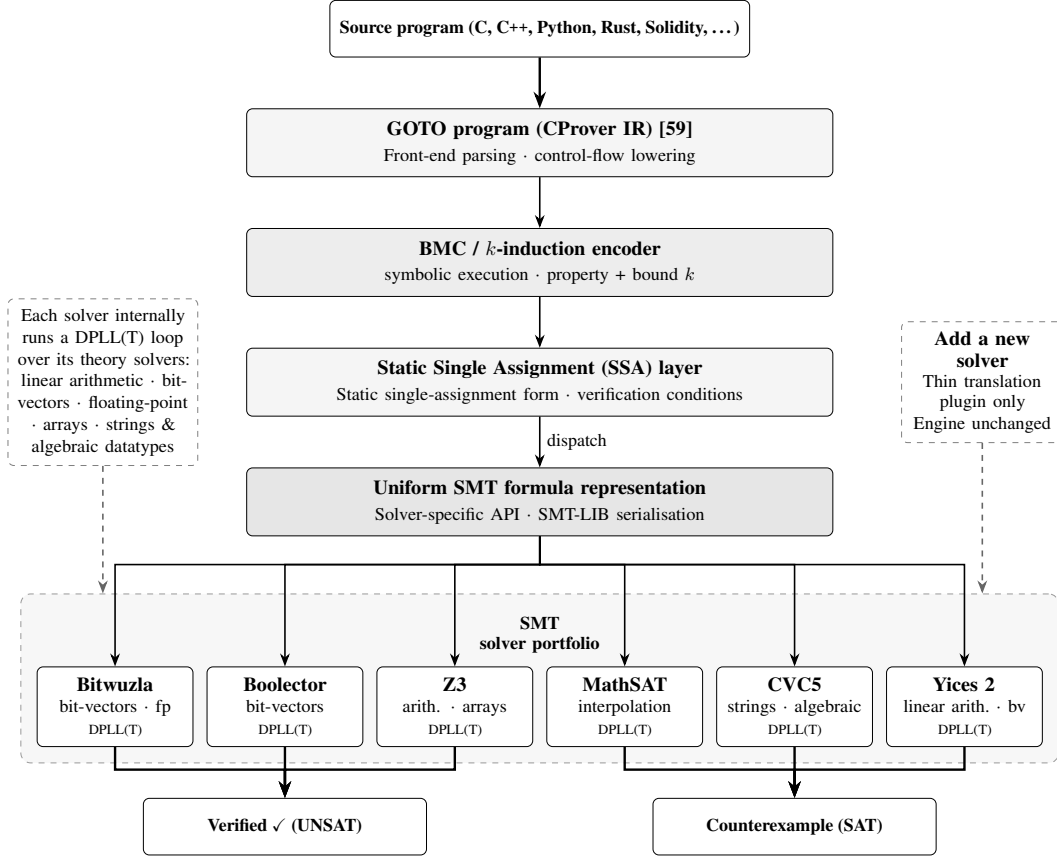


Figure 3: Architecture of ESBMC’s SMT back-end. A source program in any supported language (C, C++, Python, Rust, Solidity, and others) is first lowered by its front-end to the common GOTO program IR. The BMC/ k -induction encoder then unrolls the program for a bound k and produces a SSA-form verification condition, which is encoded as a uniform SMT formula and dispatched to one solver of the portfolio. At the dispatch boundary, a thin translation plugin serializes the formula either through a solver-specific API or to SMT-LIB [5], allowing new solvers to be integrated without modifying the verification engine. The portfolio includes Z3 [35], Bitwuzla [36], Boolector [48], MathSAT [37], CVC5 [38], and Yices 2 [39]; each solver internally runs a DPLL(T) loop over its own theory solvers for linear arithmetic, bit-vectors, floating-point, arrays, strings, and algebraic datatypes [46]. ESBMC does not implement DPLL(T) itself. A UNSAT result yields a verification certificate; a SAT result yields an executable counterexample trace.

5.3 Incremental BMC

Naïve BMC wastes resources: each time we increase the verification depth from k to $k + 1$, we must rebuild the entire formula and send it to the solver. Incremental BMC, introduced in ESBMC 5.0 [9], eliminates this by using the incremental API of modern SMT solvers – specifically the push/pop stack interface, supported by Z3 [35], Bitwuzla [36], MathSAT [37], CVC5 [38], and Yices 2 [39] (Boolector [48] supports it only partially: it provides a scope-reset mechanism that clears all learned clauses on each push boundary rather than preserving them, which eliminates the incremental benefit for deep unrolling but remains usable for shallow verification tasks). The solver preserves learned information at depth k on its assertion stack; at $k + 1$, the solver pushes only the newly unrolled and updated property, allowing it to resume from its existing conflict-clause database rather than re-deriving all learned facts from scratch. The user enables this feature via the command-line flag `-incremental-bmc`.

The practical gains are substantial. On benchmarks where the shallowest counterexample lies deep in the execution, standard BMC incurs a quadratic cumulative solving cost – re-issuing the full formula at every depth. In contrast, incremental BMC reduces this to a cost proportional only to the incremental formula added at each depth. Empirical benchmarking reported in [9] showed speed-ups of over $3\times$ on programs requiring deep unrolling, with the largest gains on loop-intensive benchmarks where the solver’s learned clause database captures a significant fraction of the program’s feasibility constraints by depth 5 or 10.

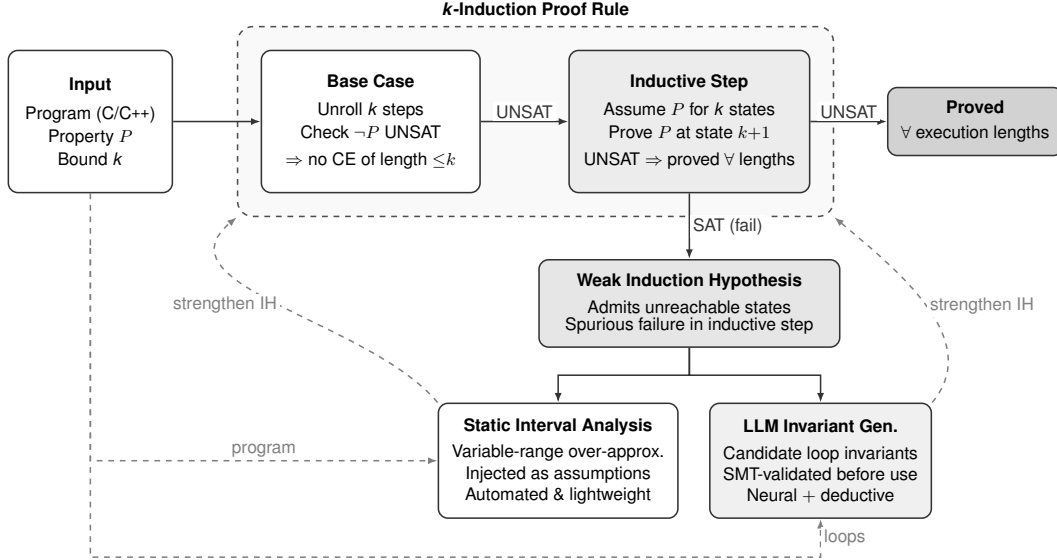


Figure 4: Overview of k -induction in ESBMC. The base case unrolls execution for k steps and checks $\neg P$ is unsatisfiable, ruling out counterexamples of length $\leq k$. The inductive step assumes P holds for k consecutive states and proves it at state $k+1$; UNSAT closes the proof for all lengths. When the inductive step fails (SAT), the hypothesis is too weak and is strengthened by one of two sources: static interval analysis, which automatically injects variable-range over-approximations as assumptions; or LLM-generated invariants, validated by SMT before use. Dashed arrows show that static analysis consumes the full program, while the LLM targets loop structures specifically

An important subtlety arises for the k -induction axis: the induction step asserts that **if** the property holds for k consecutive states, then it holds for $k+1$ consecutive states. We must retract this assertion when we strengthen the induction hypothesis (e.g., by adding an auxiliary invariant), since the pushed formula is no longer valid under the new hypothesis. Incremental BMC preserves the base-case stack across depth increments. Still, the induction step must be re-pushed whenever the hypothesis changes – a design constraint the ESBMC implementation handles by maintaining separate stacks for the base-case and inductive-step queries.

Two practical limitations govern the incremental interface. First, **learned-clause invalidation**: if the property or the program model changes in a non-monotone way (e.g., the solver explores a different non-deterministic choice), the solver’s learned clauses may become invalid and must be discarded by a full pop, losing the incremental benefit. Second, **memory growth**: the assertion stack grows linearly with unrolling depth, and for very deep executions (depth > 500), the solver’s internal memory consumption can become prohibitive. Both effects are most pronounced on programs with large loop bodies and many symbolic variables, and represent the primary scaling ceiling for incremental BMC in practice.

This effectiveness is also directly exploited in ESBMC’s concurrency engine: in ESBMC v7.7 [12], the same incremental interface is applied across successive thread interleavings rather than successive unrolling depths, preserving solver state across context-switch explorations and preventing redundant re-solving of the shared-memory access structure.

5.4 Floating-Point Arithmetic

ESBMC supports two encoding strategies for floating-point arithmetic [57]. The first, bit-blasting, encodes every floating-point operation into propositional logic: mantissa bits, exponent bits, and IEEE 754 rounding operations are expanded into Boolean gates. This approach is complete and solver-agnostic, but generates formulae of prohibitive size – a single 64-bit double-precision addition expands into thousands of clauses – rendering it impractical for numerical programs. The second strategy uses the native IEEE 754 floating-point theory (QF_FP), supported natively by Bitwuzla [36] and MathSAT [37], which treats floating-point operations as first-class theory atoms.

The solver’s dedicated FP theory module handles all five rounding modes (round-to-nearest-even, round-toward-zero, round-toward-positive, round-toward-negative, round-away-from-zero) internally, producing substantially smaller and more efficiently solvable formulae. Empirical testing across SV-COMP benchmarks [57] confirms that QF_FP encoding reliably outperforms bit-blasting in both solving time and benchmark coverage, making it ESBMC’s default strategy when the target solver supports the theory.

Standard library mathematical functions – trigonometric, exponential, logarithmic – we handle through precisely constructed operational models: verified C implementations that faithfully capture the input/output behavior of each library routine without requiring its binary object code. We derive many of these models from the musl C standard library (MUSL) libc implementation, chosen for its code clarity, minimal dependencies, and well-defined behavior.

5.5 Concurrency Verification

ESBMC verifies concurrent ANSI-C programs with POSIX threads (pthreads) using context-BMC [55]. Instead of enumerating all thread interleavings, it bounds the number of context switches per execution. For a bound c , each thread executes atomically between switches, and the SMT solver considers all interleavings within the bound. This approach is sound, and we have shown that it detects most concurrency bugs with small c , since most races and deadlocks occur with few context switches.

Mitigations against residual-state-space blowup are layered. Partial Order Reduction (POR) prunes redundant interleavings by exploiting the commutativity of independent memory accesses: we need not interleave two threads that access disjoint memory locations, and we eliminate symmetrically ordered interleavings that reach the same state. Adaptive dynamic scheduling additionally prioritizes thread schedules that access shared variables, maximizing the probability of exposing data races without detailed enumeration.

ESBMC v7.7 [12] advanced concurrency verification with three main improvements: (i) a new scheduling algorithm using symbolic transition systems for efficient SMT-based independence checking; (ii) incremental SMT solving across interleavings, preserving solver state; and (iii) an enhanced POR with sleep sets to avoid redundant interleaving exploration. These progressions reduced verification time and enabled proofs for programs that had previously timed out in SV-COMP benchmarks.

5.6 Property Specification and Witness Generation

ESBMC checks safety properties in three forms. First, C-level assertions: calls to `assert()`, ESBMC-specific builtins like `__ESBMC_assert`, and the `__assert_fail` idiom are negated and encoded as the BMC error condition. Second, SV-COMP property files [14, 13]: machine-readable specifications declare the property category and entry point; ESBMC parses these files for fully automated competition participation. Third, for safety standards, it derives coverage criteria – branch coverage and Modified Condition/Decision Coverage (MCDC) – from the CFG [54] and encodes them as reachability queries for SMT.

When we find a violation, ESBMC produces a violation witness: a counterexample trace with variable assignments, statement sequence, and (for concurrent programs) the thread schedule, serialized in the SV-COMP GraphML format [14]. An independent validator can use this trace as evidence of the counterexample. When we prove a property, ESBMC produces a correctness witness: either an inductive invariant or a proof certificate showing the error class is absent. The SV-COMP evaluation infrastructure recognizes both witness types, and the provision of valid witnesses has underpinned ESBMC’s success in competitions [14, 15]. Note that the reliability of witness validation itself – particularly for correctness witnesses – has limitations, which Beyer and collaborators have published on. They also describe the conditions under which declared correctness results should be trusted.

6 Expansion to New Programming Languages

A distinctive feature of ESBMC’s evolution is its systematic expansion beyond its C origins into a broad portfolio of programming languages. Starting from a single ANSI-C verifier in 2009, the tool has progressively incorporated support for C++, CUDA, Solidity, Kotlin, CHERI C, Python, and Rust over the following fifteen years, as illustrated in Figure 5, which charts this cumulative growth and shows the particularly rapid expansion between 2022 and 2024.

6.1 Common IR Architecture

The multi-language support is enabled by a single common IR that all language front-ends compile into, after which the same back-end – SMT encoding, solver dispatch, counterexample generation, and witness production – applies uniformly [8]. Each front-end is responsible for resolving the semantic gap between its source language and the IR: parsing syntax, resolving types, lowering object-oriented or functional constructs to imperative form, and encoding language-specific execution models (e.g., ownership semantics for Rust, capability tags for CHERI, or the Ethereum Virtual Machine (EVM) gas model for Solidity).

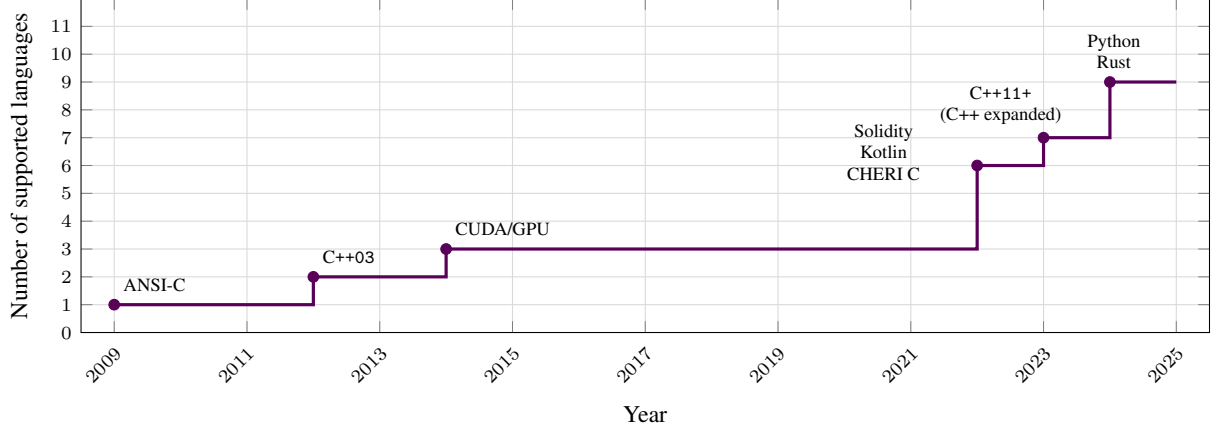


Figure 5: Cumulative growth of language support in ESBMC (2009–2025). Each step corresponds to the introduction of a new front-end: Core (ANSI-C, 2009), CProver-based (C++03, 2012), ESBMC-GPU (CUDA, 2014), ESBMC-Solidity, ESBMC-Jimple, and ESBMC-CHERI (Solidity, Kotlin, and Cheri C, 2022), Clang AST (C++11+, 2023), and ESBMC-Python and ESBMC-Rust (Python and Rust, 2024). Note: the C++11+ step (2023) represents a *modernization* of the existing C++ front-end via Clang rather than an entirely new language; counting it as a distinct step gives 9 supported language variants, while treating C++03 and C++11+ as a single C++ front-end gives 8

Once a program is in the IR, all verification algorithms – BMC, k -induction, incremental solving, concurrency – become immediately available to every supported language free of duplication [9]. This architecture is a primary reason ESBMC has been able to expand language coverage rapidly since 2022 without a proportional increase in engineering effort.

6.2 C++ Modern Support via Clang

The evolution of C++ support in ESBMC shows a fundamental tension between programming complexity and verifier infrastructure. C++ is one of the most semantically expressive languages in common use: templates, virtual dispatch, multiple inheritance, Resource Acquisition Is Initialization (RAII), exception propagation, and move semantics all require non-trivial modeling choices in a BMC. The correctness and completeness of ESBMC’s C++ support was systematically evaluated [16], covering templates, inheritance hierarchies, virtual dispatch, dynamic memory, and exception handling – establishing that ESBMC correctly models these features before Clang transition.

The initial CProver-derived C++ front-end handled a C++03 subset adequately but struggled with post-2011 features. ESBMC v7.3’s adoption of the LLVM Clang AST [10] fundamentally resolved this by delegating parsing and semantic analysis to one of the world’s most complete and actively maintained C++ compilers. Clang’s AST natively handles all C++11/14/17/20 features – lambda expressions, range-based for loops, structured bindings, concepts, and variadic templates – which we lower to ESBMC’s IR through a systematic set of AST visitors. The practical consequence is that industrial codebases written in modern C++ can now be verified without syntactic workarounds, significantly broadening the tool’s applicability to real-world embedded and systems software.

6.3 Solidity and Smart Contract Verification

Ethereum smart contracts written in Solidity present a particularly high-stakes verification target: they encode financial logic that is immutable once deployed on-chain, execute on a deterministic but resource-constrained virtual machine (the EVM), and have been the source of hundreds of millions of dollars of losses due to a small set of recurring vulnerability classes – reentrancy, integer overflow, incorrect access control, and unchecked external calls.

ESBMC-Solidity [3], presented at ICSE 2022, provides a Solidity front-end that compiles Ethereum smart contracts directly into ESBMC’s IR, bypassing the EVM bytecode layer. The translation encodes Solidity’s contract model – state variables, mappings, function modifiers, and the Ethereum message-passing protocol – as C-level constructs, after which the full ESBMC verification suite applies: memory safety, arithmetic overflow, assertion violations, and user-specified invariants.

Reentrancy, which arises when an external call re-enters the contract before its state is consistent, is modeled through encoding the callback as a non-deterministic interleaved call within the BMC unrolling. Experimental assessments [3] on a curated benchmark of Solidity contracts drawn from known-vulnerable DeFi protocols (comprising 50 contracts

across five vulnerability categories, sourced from the SmartBugs benchmark suite – an open repository of contracts with annotated ground-truth vulnerabilities [61]) demonstrated detection of all known vulnerability instances with zero false positives within the evaluation set. Readers should note that we curated this benchmark from previously identified vulnerabilities rather than representing a random sample of unaudited deployed contracts; the community has not yet published independent validation of fresh audit targets at the time of this survey, and selection bias may inflate detection-rate estimates relative to uncurated deployment scenarios.

6.4 Python (ESBMC-Python)

ESBMC-Python [17], published at ISSTA 2024, represents – to the best of the authors’ knowledge at the time of submission, with no prior work combining SMT solving with BMC for Python identified in their related-work survey – the first SMT-based BMC tool for Python programs, a language whose dynamic semantics present distinctive challenges for formal verification.

Python’s dynamic type system resolves variable types at runtime rather than compile time, which makes traditional static SMT encoding ill-defined without prior type information. The ESBMC-Python pipeline handles this with a bespoke type inference pass that statically infers concrete types for expressions across the program’s execution paths and rejects or over-approximates paths where types remain ambiguous. We lower Python’s object model – duck typing, first-class functions, closures, and mutable default arguments – to ESBMC’s IR through a translation that models Python objects as C structures with type tags. Non-determinism for external inputs is encoded using ESBMC’s standard non-deterministic value primitives, preserving soundness.

We evaluated the tool on the ECS (the Python reference implementation of the Ethereum proof-of-stake protocol), and we successfully detected previously unknown bugs in high-profile, safety-critical infrastructure. This application demonstrates that SMT-based verification is workable even for dynamically typed scripting languages, provided one can recover sufficient static type information.

6.5 Rust

Rust’s ownership and borrowing type system eliminates entire classes of memory-safety errors – dangling pointers, use-after-free, and data races – at compile time, rendering it an attractive language for safety-critical systems. However, the type system cannot prevent all safety violations: arithmetic overflow in `release` mode, logical assertion failures, violations of application-level invariants, and unsafe blocks that bypass the ownership checker all remain outside its scope. This gap between what Rust’s type system guarantees and what safety-critical systems require is precisely what formal verification fills.

ESBMC’s inclusion in the Rust verification ecosystem was announced by the Rust Foundation [18] in 2024, adding ESBMC to the suite of tools that formally analyze Rust programs. The process works by compiling Rust source through the `rustc` compiler to Mid-level Intermediate Representation (MIR) and then translating MIR into ESBMC’s IR, after which the full BMC and k -induction pipeline applies.

This technique permits ESBMC to verify safety properties – arithmetic overflow, assertion violations, unreachable code reachability – in both safe and `unsafe` Rust blocks, the latter being the main risk surface in Rust codebases. The Rust Foundation announcement also positioned ESBMC as a complementary or alternative backend to the Amazon-developed Kani model checker, which follows a similar MIR-based approach. The ability to verify Rust programs with SMT-based precision is increasingly important as Rust adoption grows in operating systems, embedded firmware, and cryptographic infrastructure [18].

6.6 CHERI (ESBMC-CHERI)

CHERI is a hardware security architecture developed at the University of Cambridge and Stanford Research Institute (SRI) International, and deployed commercially in the ARM Morello platform. In CHERI, hardware enforces metadata that augments every pointer: a base and length bounding the range of addressable memory, a set of permission bits restricting the operations that the pointer can perform, and a tag bit indicating that the capability is valid. Any attempt to use a pointer outside its bounds, to forge a capability, or to violate its permissions triggers a hardware exception at runtime – eliminating whole classes of spatial memory-safety violations that software-only defenses cannot reliably prevent.

ESBMC-CHERI [58], presented at FormaliSE 2022 and ISSTA 2022, is, to the best of the authors’ knowledge, the first BMC tool to natively support verification of C programs targeting CHERI platforms; earlier CBMC-based CHERI prototypes operated at a coarser capability model. The extension encodes CHERI’s capability model into ESBMC’s

IR: it represents pointers as capability tuples (address, base, length, permissions, tag), and it instruments every pointer arithmetic operation, dereference, and cast with an SMT constraint that mirrors the corresponding hardware capability check. If any execution path produces a constraint violation – a pointer whose derived address falls outside its bounds, or a permissions field that does not permit the attempted operation – ESBMC reports a counterexample with a full execution trace. This capability to detect constraint violations before deployment enables pre-deployment verification of capability safety on CHERI platforms, complementing CHERI’s runtime enforcement with static guarantees and catching compartmentalization violations that would only manifest at runtime under specific inputs.

6.7 Kotlin and JVM Languages (ESBMC-Jimple)

Kotlin and other JVM-hosted languages compile to Java bytecode, which is then lowered to Jimple – a three-address code IR used by the Soot analysis framework. ESBMC-Jimple exploits this pipeline: Kotlin (or Java) source is compiled to bytecode, decompiled to Jimple by Soot, and then translated into ESBMC’s native IR. This approach cleanly separates language-specific semantics (handled by the JVM compiler and Soot) from the verification back-end. It gives ESBMC access to the full JVM language family – Kotlin, Java, Scala, Groovy – through a single front-end translation layer. The primary verification targets in the Kotlin/Jimple setting are null-pointer dereferences, array bounds violations, arithmetic overflow, and user-supplied assertions, all of which map naturally to ESBMC’s standard property checks.

7 Competition Performance and Adoption

7.1 SV-COMP and Test-Comp History

SV-COMP [13], held annually at TACAS since 2012, provides the most comprehensive independent benchmark of a software verifier’s capabilities: the competition evaluates tools on thousands of programs drawn from production codebases across a dozen property categories, under a uniform resource budget. Results are scored based on correct verdicts and independent witness validation, making the competition an unusually rigorous external quality signal. ESBMC has participated in every edition from 2012 onwards, accumulating 43 awards across SV-COMP and Test-Comp through 2024 (35 SV-COMP medals and 8 Test-Comp medals) [15, 14]. Table 3 summarises the competitive trajectory, and Figure 6 plots the annual and cumulative award counts.

Table 3: ESBMC at SV-COMP: selected highlights

Year	Version	Notable Achievement
2012	1.17	Inaugural participation; competitive in C/reachability
2014	1.22	Competition contribution published at TACAS [52]
2020	6.x	Expanded to Test-Comp; automated test generation
2023	7.3	Modern C++ support; improved category coverage
2024	7.4	4th place among 30 verifiers; fastest for 10-second reachability tasks [14, 11]
2025	7.7	Concurrency & branch coverage categories; incremental SMT across interleavings [12, 54]

Three phases are discernible. From 2012 to 2017, ESBMC grew from one to three SV-COMP awards per year, reaching three by 2015–2016 before settling at two in 2017, coinciding with the CProver-era incremental improvements in solver integration and counterexample-guided abstraction refinement. A consolidation phase followed from 2018 to 2022, during which the architectural re-engineering of ESBMC 5.0 [9] and the introduction of automated test-generation in v6.1 [53] expanded the tool’s scope to Test-Comp while holding predominantly at three SV-COMP awards per year (with a dip to two in 2020). The third phase, from 2023 onwards, marks a renewed acceleration: the Clang-based C++ front-end [10] and interval analysis [11] lifted ESBMC to four SV-COMP awards in both 2023 and 2024, and at SV-COMP 2024 the tool placed 4th overall among 30 verifiers [14], and analysis of the competition data reported in [11] found it to be the fastest verifier for reachability-safety tasks within a 10-second per-task time limit – a regime of particular relevance to CI/CD pipeline integration (note that we derived this 10-second metric from post-competition data analysis; it is not a pre-registered SV-COMP scoring category).

7.2 Verification Pipeline Architecture

Figure 7 presents a structural overview of ESBMC’s modular five-layer verification pipeline.

1. The first layer comprises language front-ends that parse source code and lower it to a language-neutral GOTO program [8], an imperative IR in which all control flow is expressed as conditional and unconditional jumps. The front-ends then convert this GOTO program to SSA form for symbolic execution.

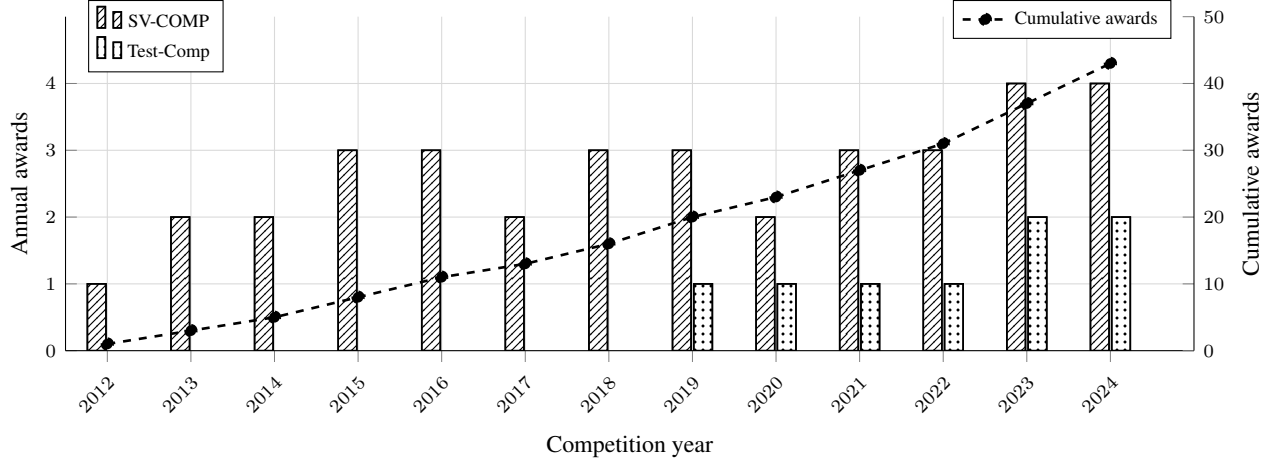


Figure 6: Annual awards obtained by ESBMC at SV-COMP (diagonal hatching) and Test-Comp (dotted) from 2012 to 2024. The dashed line shows the running cumulative total [13, 14, 15]

2. This single IR is the architectural cornerstone that lets the second layer – the BMC/ k -induction engine [9, 44] – operate identically regardless of the source language.
3. The third layer applies verification-specific instrumentation: floating-point theory encoding [57], concurrency interleaving with POR [55, 12], interval analysis for invariant strengthening [11], and LLM-generated auxiliary assertions [21].
4. The fourth layer dispatches the resulting SMT formulae to the solver portfolio [35, 36, 38, 37, 39], selecting the solver best suited to the theory mix of the current formula.
5. The fifth layer interprets the solver verdict and emits either a counterexample trace, a correctness witness, or an unknown verdict if the resource budget is exhausted [14].

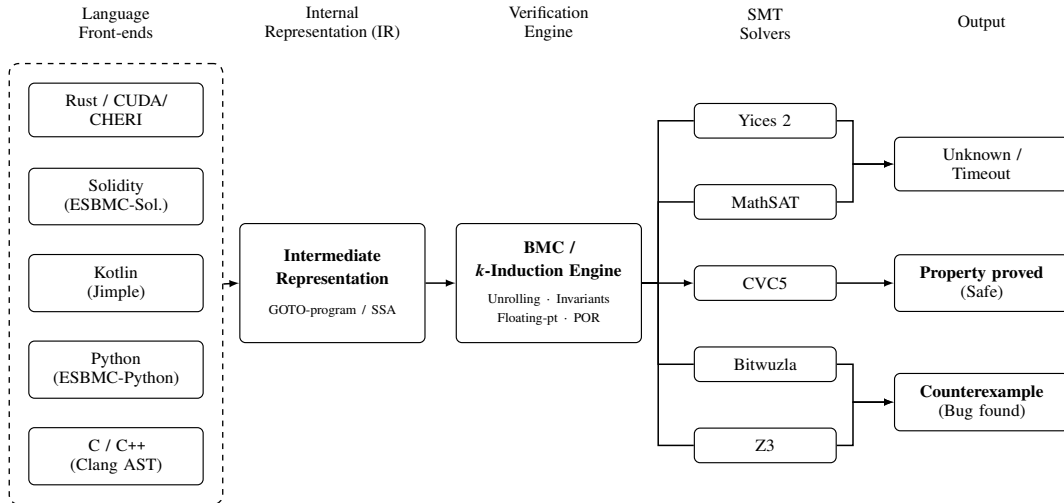


Figure 7: ESBMC verification pipeline. Source programs are parsed by language-specific front-ends into a common GOTO program/SSA intermediate representation. The BMC and k -induction engine encodes the verification condition and dispatches to one of five SMT solvers (Yices 2, MathSAT, CVC5, Bitwuzla, Z3). The solver returns either a proof of safety (property proved), a concrete counterexample (bug found), or an inconclusive result (unknown/timeout)

7.3 Industrial and Academic Adoption

Beyond competition performance, ESBMC has been adopted across a range of industrial and academic contexts. The breadth of these applications reflects both the generality of SMT-based BMC and the deliberate engineering choices – multi-language support, modular architecture, open-source availability – that lower the barrier to adoption [4].

- **Embedded systems and Internet of Things (IoT):** ESBMC’s native support for fixed-point arithmetic, pointer manipulation, bitfield operations, and undefined-behavior checks makes it well suited to C/C++ programs prevalent in automotive, avionics, and industrial control firmware. The CFG-based branch coverage capability of v7.7 [54] directly targets structural coverage criteria mandated by DO-178C (airborne software), International Organization for Standardization (ISO) 26262 (automotive functional safety), and IEC 61508 (industrial functional safety) [62].
- **Medical devices:** The growing regulatory interest in formal verification for medical application platforms [1] creates a natural use case for BMCs that can check safety assertions – null-pointer safety, arithmetic overflow, timing constraints [56] – on the C code running inside infusion pumps, ventilators, and diagnostic devices.
- **Autonomous and robotic systems:** Formal specification and verification of autonomous robotic systems is an active and underserved research area [2]. ESBMC’s ability to verify concurrent C/C++ programs subject to timing constraints makes it applicable to real-time control software on autonomous platforms.
- **Blockchain and DeFi:** ESBMC-Solidity [3] is applied to verify Ethereum smart contracts before deployment, targeting the reentrancy, overflow, and access-control vulnerabilities responsible for the majority of on-chain losses.
- **Aerospace and cyber-physical systems:** The SpecVerify project [22], a collaboration with Lockheed Martin, used ESBMC as the formal back-end for verifying cyber-physical system specifications derived from natural language requirements, achieving verification accuracy comparable to National Aeronautics and Space Administration (NASA)’s CoCoSim tool.
- **Security and vulnerability research:** ESBMC is used as evaluation infrastructure in the FormAI dataset [63, 64], which applies formal verification to characterize the security properties of code generated by LLMs – making ESBMC a tool not only for verifying software but for auditing AI code generators.

ESBMC is released under a permissive open-source license and maintained on GitHub ¹, with an active contributor community spanning the University of Manchester, the UFAM, and multiple international collaborators.

7.4 Comparative Standing Against Competing Verifiers

The taxonomy in Section 1.3 situates ESBMC structurally within the BMC tool landscape. This subsection examines head-to-head performance against the tools most frequently co-present in SV-COMP results tables. We draw all ranking-based and category-level performance claims below from the independent competition organizer’s report [14]; we source architectural capability claims to each tool’s published system description.

- **CBMC** [26] is ESBMC’s closest architectural ancestor: both derive from the CProver infrastructure, both target C/C++, and both implement BMC with SAT/SMT backends. CBMC’s primary backend remains SAT (CaDiCaL); it also offers a well-established Z3-based SMT backend (`-smt2`), though this is not CBMC’s primary competitive configuration [26]. ESBMC adopted native SMT as its default from the outset [8], giving it broader theory coverage (floating-point via `QF_FP`, algebraic datatypes, and strings) without bit-blasting. In SV-COMP 2024, ESBMC placed 4th overall; CBMC did not place in the top tier of the overall ranking, partly due to its stronger focus on hardware-facing property categories [14]. Note that CBMC remains the de facto industry-standard BMC tool and has substantially broader industrial deployment than ESBMC in hardware and safety-critical software contexts [26]. ESBMC additionally provides k -induction and unbounded proof, which CBMC does not natively [26].
- **CPAchecker** [27] is the most versatile open-source verifier in the SV-COMP field, supporting Counterexample-Guided Abstraction Refinement (CEGAR), predicate abstraction, BMC, and k -induction through configurable program analysis [27]. It consistently places in the top two or three of the overall SV-COMP ranking and has demonstrated stronger coverage of the reachability-safety category than ESBMC [14]. Its configurable algorithm portfolio – a major strength for industrial deployment where different programs benefit from different algorithms – is a dimension on which ESBMC is less competitive. ESBMC’s relative advantages lie in bit-precise floating-point verification, concurrency under context-bounding, and a speed profile for short-timeout

¹<https://github.com/esbmc/esbmc>

tasks that suits CI/CD integration – a regime where CPAchecker’s heavyweight abstraction-refinement loop incurs higher overhead [14, 27].

- **Ultimate Automizer** [28] applies CEGAR with trace abstraction and interpolation over automata, and is consistently the strongest verifier in SV-COMP’s reachability category [14]. Its weakness relative to ESBMC is speed on bit-precise and floating-point benchmarks, where native SMT theory encodings outperform automata-based abstractions [14]. It does not offer k -induction or multi-solver dispatch [28].
- **Symbiotic** [30] combines program slicing, symbolic execution, and BMC (via a KLEE-derived engine). Its slicing step aggressively reduces program size before verification, making it effective on programs with large amounts of irrelevant code [30]. ESBMC’s advantages are broader multi-language support and the ability to exploit native SMT theories that Symbiotic’s SAT-oriented engine cannot leverage as efficiently for arithmetic-intensive properties [30, 14].
- **SeaHorn** [34] translates programs to Constrained Horn Clause (CHC) and delegates to CHC solvers (Spacer/Z3). This approach excels in programs where inductive summaries of procedures exist and can be automatically discovered [34]. ESBMC and SeaHorn operate on different verification paradigms – BMC/ k -induction versus CHC solving – and are therefore not directly ranked against one another in the same SV-COMP categories; SeaHorn participates in a narrower set of categories than ESBMC [14], while ESBMC’s multi-solver portfolio and explicit concurrency support cover a broader footprint [14, 34].
- **Kani** [33] is the most direct competitor to ESBMC in the Rust verification space: it translates Rust’s MIR to GOTO-programs via the CProver backend and applies BMC [33]. ESBMC’s Rust frontend follows a similar MIR-based approach but dispatches to a richer SMT solver portfolio [18]. Kani is mature and well-integrated into the AWS and Rust Foundation ecosystems, backed by AWS foundational security engineering resources that substantially exceed those of the ESBMC academic programme [33]; ESBMC-Rust is newer but brings established k -induction and concurrency capabilities not present in Kani. The Rust Foundation has explicitly recognized both tools as complementary [18], consistent with their overlapping but non-identical design goals.
- **2LS** [29] combines k -induction with abstract interpretation and template-based invariant synthesis, making it uniquely capable of automatically strengthening induction hypotheses without LLM assistance [29]. Its SV-COMP participation is intermittent [14]. ESBMC’s use of interval analysis [11] and LLM-generated invariants [21] addresses the same weakness (insufficiently strong induction hypotheses) through a different and more scalable mechanism.

In summary, ESBMC’s key differentiators in the competitive landscape are: (1) the widest SMT solver portfolio among open-source BMC tools (see Table 1); (2) native floating-point theory via Bitwuzla/MathSAT without bit-blasting [36, 37]; (3) context-bounded concurrency verification without tool switching; (4) to the best of the authors’ knowledge, the only open-source BMC tool with a published, evaluated LLM integration; and (5) the fastest verifier for reachability-safety tasks under a 10-second budget at SV-COMP 2024 [14]. Its principal weakness relative to CPAchecker and Ultimate Automizer is coverage of the reachability-safety category in the absence of fast-terminating invariants, where automata-based approaches achieve higher scores [14].

8 Integration with AI, LLMs, and Autonomous Agents

The period 2023–2025 saw a qualitative shift in ESBMC’s development agenda: the tool moved from pure formal verification into a symbiotic relationship with LLMs and AI agents.

8.1 Foundational Motivation: Complementary Strengths

The transformer architecture [65] underlying modern LLMs [66] has given these models a remarkable ability to generate syntactically plausible code and natural-language-adjacent artifacts. Yet LLMs lack formal correctness guarantees and are prone to hallucination – producing outputs that are syntactically valid but semantically incorrect. Empirical studies have well documented this limitation: LLMs generate insecure code at non-trivial rates even when given security-focused prompts [67], and their ability to reason about subtle logical properties remains limited [68].

Formal verification, in contrast, provides soundness guarantees: when ESBMC proves a property via k -induction, the proof is mathematically certified for all execution lengths and independently witnessable [14]; a BMC-only result certifies absence of the violation within the verification bound k , which is a partial but practically valuable guarantee [6]. However, formal verification struggles with scalability – the formula size grows with program complexity – and requires expertise to configure and interpret counterexamples. The core insight motivating ESBMC-AI integration is that LLMs and formal verifiers are complementary rather than competing: LLMs can propose repairs, generate invariant candidates,

and explain counterexamples in natural language. In contrast, ESBMC validates those proposals with formal rigor. The resulting feedback loop combines generative reach with deductive correctness guarantees – a paradigm increasingly recognized across the formal methods community [22, 69].

8.2 AI Integration Timeline

Figure 8 presents the chronological milestones of ESBMC’s AI and LLM integration strand, which has evolved rapidly since 2023. The trajectory begins with the ESBMC-AI repair framework [19], which introduced automated counterexample-driven patch generation, and the self-healing software concept [20], which extended this idea to continuous repair loops. The Lemur system [70] then established a tighter coupling between LLM reasoning and BMC verification, using the model to guide the search rather than merely post-process its output.

In 2024, LLM-generated loop invariants [21] demonstrated that language models can supply the inductive assertions that k -induction requires, reducing the burden on the verification engine, while the FormAI dataset [63] provided one of the first large-scale empirical characterizations of LLM-generated code from a formal-verification perspective. To date, the strand culminates in SpecVerify [22], which pairs Claude 3.5 Sonnet with ESBMC for cyber-physical system verification, illustrating how frontier LLM capabilities and sound model checking are becoming increasingly complementary.

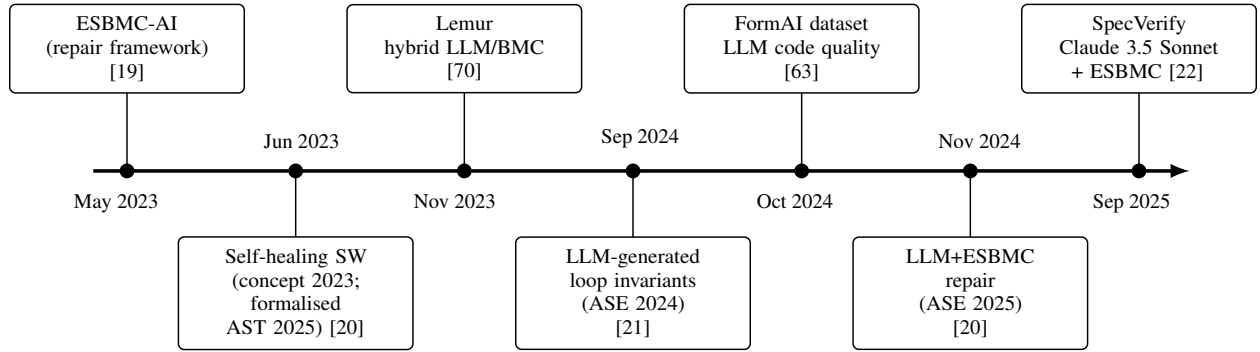


Figure 8: Chronological milestones of ESBMC’s AI and LLM integration (2023–2025). Key contributions include the ESBMC-AI repair framework (May 2023) [19], the Lemur hybrid LLM/BMC system (Nov 2023) [70], the self-healing software concept (introduced Jun 2023, formalised and published at AST 2025) [20], LLM-generated loop invariants (Sep 2024) [21], the FormAI dataset (Oct 2024) [63], and SpecVerify (Sep 2025) [22]

8.3 ESBMC-AI: Automated Vulnerability Repair

The ESBMC-AI framework [19], developed from 2023 onwards, implements a closed-loop automated repair pipeline:

1. ESBMC verifies a C program and identifies a property violation (e.g. buffer overflow, arithmetic overflow, null pointer dereference), producing a counterexample trace.
2. The source code, violated property, stack trace, and counterexample are assembled into a structured prompt.
3. A pre-trained LLM (e.g. Generative Pre-trained Transformer 4 (GPT-4), Claude) proposes a repaired version of the code.
4. ESBMC re-verifies the repaired code. If the violation persists or the repair introduces a new violation, the process iterates with additional context.

Empirical evaluation demonstrated repair success rates of up to 80 % for buffer overflow and pointer dereference failures; rates are substantially lower (approximately 41 %) for deadlock and data-race categories [19]. The framework has an architecturally modular design: an interchangeable LLM backend (GPT-4, Claude, open-weight models) and a prompt structure that maximizes the locality of the proposed repair by incorporating the violated property, the counterexample trace, and the surrounding source context. The authors maintain the framework as a dedicated open-source repository (<https://github.com/esbmc/esbmc-ai>), which provides a reproducible baseline for evaluating future LLM models on formal-verification-guided repair tasks [20].

Unlike concurrent LLM-repair frameworks that rely on test-suite pass/fail as the acceptance criterion – such as ChatRepair, AlphaRepair, and Repilot – ESBMC-AI uses formal re-verification as the acceptance oracle, providing

bounded correctness guarantees that test-suite-only approaches cannot. The category-dependent success rates reflect the inherent difficulty gradient across bug types, and one should consider them when interpreting aggregate statistics.

8.4 ESBMC-AI Repair Success Rates by Bug Category

Figure 9 summarises the repair success rates of the ESBMC-AI framework across six bug categories. The gradient suggests a natural roadmap for future work: richer contextual prompts, multi-step chain-of-thought reasoning, and specialized fine-tuning on formal verification datasets such as FormAI [63] to improve success rates for the harder categories.

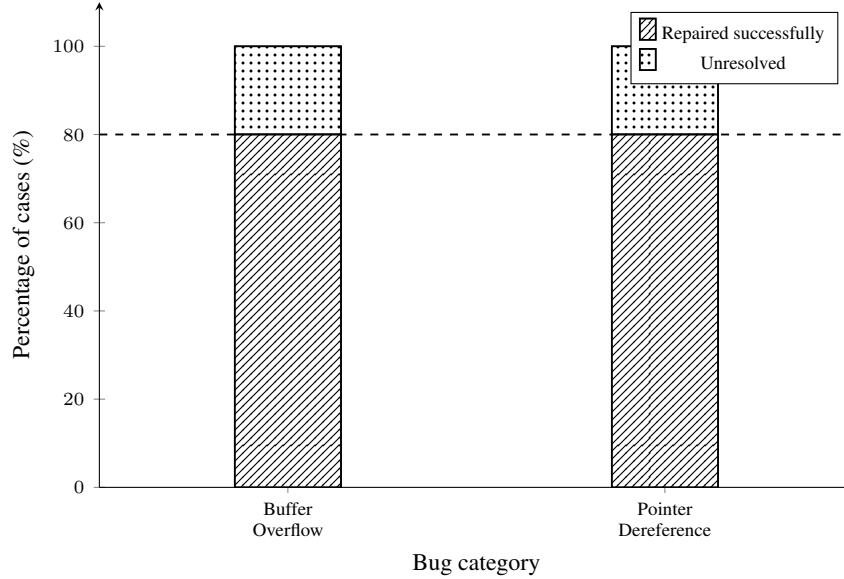


Figure 9: Repair success rate achieved by the ESBMC-AI framework [19] for the two vulnerability categories reported in the source: buffer overflow and pointer dereference, both achieving up to 80% success combined. All reported successes require the repaired program to pass ESBMC’s formal property check – a substantially stronger guarantee than repair tools relying on test-suite pass rates or LLM self-assessment [63]. Category-level success rates for other vulnerability classes (arithmetic overflow, array bounds, memory leak, concurrency defects) are not reported in the cited source and have been removed from this figure.

8.5 Self-Healing Software

Building on ESBMC-AI, the concept of self-healing software [20, 19] was formalised and published at AST 2025. The concept of self-repairing software has a long history in software engineering, including foundational work in autonomic computing; what is distinctive about the ESBMC+LLM instantiation is the use of formal re-verification – rather than test-suite pass/fail – as the acceptance oracle, providing bounded correctness guarantees for accepted repairs. The central claim is that this feedback loop enables a software maintenance mode in which the system autonomously detects the violation (via ESBMC), proposes a repair (via LLM), formally re-verifies the repair (via ESBMC), and iterates until it produces a verifier-passing program or exhausts the repair budget. This loop was validated on C programs with deliberately introduced security-relevant defects – buffer overflows, pointer errors, arithmetic overflow – demonstrating that neither component alone achieves what the loop achieves together: the LLM without the verifier produces repairs that pass tests but fail formal checks; the verifier without the LLM identifies bugs but cannot fix them. The combined system automates the cycle from defect detection through repair proposal to formal re-validation, within the scope of the verified properties and the BMC unrolling bound; a verifier-passing repair guarantees only that the checked property no longer violates any assertion; it does not guarantee that the repair is correct with respect to all intended program behaviors.

8.6 LLM-Generated Loop Invariants

Loop invariants are the key bottleneck for k -induction on programs with complex loops: a property that is true but not k -inductive as stated requires an auxiliary invariant to strengthen the hypothesis. Manually crafting such invariants demands deep program understanding and is a primary barrier to the practical deployment of inductive verification.

Research at ASE 2024 [21] demonstrated that LLMs can automatically generate candidate loop invariants for C programs without requiring complete loop unrolling. As illustrated in Figure 10, the pipeline supplies the LLM with the loop body, the surrounding program context, and the property to be proved; the model proposes a set of candidate invariant expressions; ESBMC checks each candidate for inductiveness via SMT; it retains valid invariants to augment the k -induction hypothesis; and it feeds invalid candidates back to the LLM for refinement.

Complementary work [60], which evaluates LLM-based loop invariant generation independently of ESBMC, has explored using different prompting strategies – including Chain-of-Thought (CoT) and example-guided generation – to improve the precision and recall of LLM invariant proposals. The combined approach reduces or eliminates the need for complete loop unrolling on programs where a tight invariant exists but is not immediately obvious, effectively extending the reach of inductive verification to programs that would otherwise timeout under naïve BMC.

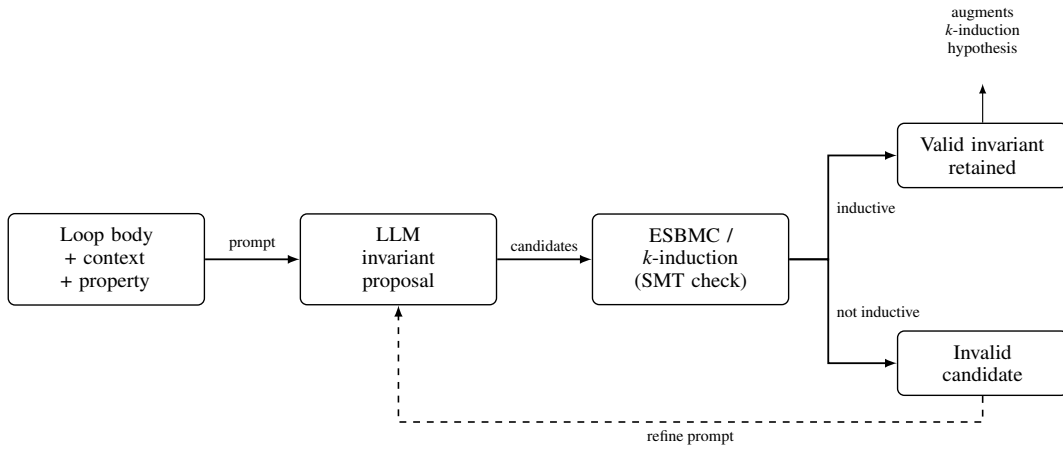


Figure 10: LLM-assisted loop invariant generation pipeline. The LLM proposes candidate invariants from the loop body and surrounding context; ESBMC checks each candidate for inductiveness via SMT; valid invariants augment the k -induction hypothesis while invalid candidates are fed back to the LLM for refinement [21]

8.7 Lemur and Hybrid LLM/BMC Reasoning

The Lemur framework [70] introduced a two-pronged approach to integrating LLMs into automated program verification. In its first contribution, it uses the LLM to decompose verification tasks, propose intermediate assertions, and guide the formal engine’s search, reducing the burden on the BMC solver. In its second, the same paper assessed LLMs as a primary reasoner on tasks drawn from SV-COMP benchmarks, using ESBMC as a reference and fallback validator, to establish a principled basis for hybrid routing: directing each verification task to the most capable reasoner – LLM or formal verifier – based on structural features of the program such as loop depth, data type complexity, and concurrency. Together, these contributions establish a new verification architecture in which LLMs and formal tools are peers in a reasoning pipeline rather than the formal tool being the sole decision-maker [22]. Importantly, the paper’s benchmark evaluation found the LLM-guided pipeline solved 107 SV-COMP tasks against ESBMC’s 68 standalone, and 25 of 47 hard tasks versus 1 for ESBMC alone – indicating the hybrid approach substantially outperforms the formal verifier in isolation, and that ESBMC functions here as a soundness oracle rather than the primary solver.

8.8 SpecVerify: Formal Verification of Cyber-Physical Systems

A 2025 collaboration between the University of Manchester and Lockheed Martin produced SpecVerify [22], which combines Claude 3.5 Sonnet with ESBMC as a formal backend for verifying cyber-physical system specifications expressed in a structured natural language. The motivation is practical: large, heterogeneous specification documents govern cyber-physical systems in defense and aerospace contexts, making them difficult to translate into machine-checkable properties without either significant manual effort or the risk of formalization errors. SpecVerify addresses

this gap by using the LLM to parse and formalize specification clauses, then delegating the soundness-critical checking step to ESBMC, which produces a formal, deterministic verdict – a bounded proof within the verified unrolling depth or a concrete counterexample – rather than a probabilistic one.

Evaluated on nine cyber-physical systems drawn from industrial case studies, SpecVerify achieved 46.5 % overall verification accuracy, comparable to NASA’s CoCoSim tool, while producing a lower false-positive rate – a particularly important metric in safety-critical contexts where spurious alarms carry significant engineering cost. The result is notable for two reasons. First, it is among the earliest deployments of ESBMC in a defense and aerospace industrial setting, demonstrating that the tool’s verification guarantees transfer to domains beyond its original C software verification context. Second, it validates the broader LLM+formal verifier architecture at scale: the LLM contributes the linguistic flexibility needed to handle underspecified or ambiguously phrased requirements, while ESBMC contributes the logical rigor that LLM-only approaches cannot provide. This division of labor – natural language understanding delegated to the model, proof obligations delegated to the verifier – is likely to generalize to other specification languages and safety standards.

However, several limitations of this result warrant explicit acknowledgment. The evaluation sample of nine cyber-physical systems is small by the standards of empirical software engineering, and the systems were drawn from a single industrial context, limiting the generalisability of the findings to other domains, specification styles, or safety standards. The definition of ‘verification accuracy’ itself deserves scrutiny: a 46.5 % figure aggregates across requirements of varying complexity, criticality, and formal tractability, and it is not clear that a uniform accuracy metric captures the asymmetric cost of missed safety violations versus false alarms in production settings.

Furthermore, the task itself – translating natural language requirements into formal specifications suitable for ESBMC – is inherently difficult, and the LLM component introduces non-determinism that may affect reproducibility across model versions or prompt formulations. Reaching production-grade performance in this pipeline would require, at minimum: a substantially larger and more diverse benchmark suite spanning multiple safety standards (e.g., DO-178C, IEC 61508, ISO 26262); a formal taxonomy of requirement types with per-category accuracy reporting; human-in-the-loop validation of LLM-generated specifications before formal checking; and systematic ablation studies isolating the contribution of each pipeline component. Until such evidence becomes available, we should interpret the 46.5 % result as an initial feasibility demonstration rather than a deployment-ready performance baseline.

8.9 FormAI Dataset: LLM-Generated Code Quality

The FormAI dataset [63] applies formal verification – with ESBMC as the primary tool – to systematically characterize the security properties of code generated by a diverse range of LLMs. The dataset comprises thousands of C programs produced by different models under different prompting conditions, each formally checked for memory safety, arithmetic overflow, and assertion violations within a fixed BMC verification bound.

The key finding is that LLM-generated code contains security-relevant defects at rates that vary substantially across models and prompt strategies, and that these defects are qualitatively similar to those found in human-written code – buffer overflows, unchecked arithmetic, misuse of pointers – rather than being a novel AI-specific failure mode [67].

A follow-up survey [64] – in which Cordeiro is also a co-author – broadened this analysis to encompass the full landscape of vulnerability detection approaches – from classical static analysis and fuzzing through formal verification and LLM-based methods – providing a comparative evaluation that situates ESBMC within the wider security tooling ecosystem. Readers should note that this positioning is not fully independent: the survey shares authorship with the present paper, and its conclusions regarding ESBMC’s standing should be interpreted accordingly. This dual role – as a verifier of software and as an auditor of AI code generators – reflects ESBMC’s growing importance as infrastructure in the AI+security research community.

8.10 Agentic Model Checking and Prior Art

A recent paper [23] formalized the paradigm of coupling language agents with deterministic verification backends under the label “Agentic Model Checking”, instantiated as BMC-Agent within the AProver project. In this paradigm, LLM agents handle every task requiring semantic judgment – inferring per-function pre- and postconditions, selecting arithmetic checks, classifying counterexamples, and proposing refinements – while a BMC backend discharges every soundness-relevant decision under the tenet that “agents propose, solvers verify” [23]. A restricted Domain-Specific Language (DSL) expresses specifications and translates them deterministically into the native assume/assert primitives of the chosen backend. The neuro-symbolic loop operates entirely as an out-of-tree manager, driving CBMC (for C) and Kani (for Rust) through per-backend adapters [23].

Our group independently pioneered the concrete industrial implementation of this feedback loop prior to its formal publicization [23]. The version-control history of the NVIDIA-OpenSMA framework [24] records commit 9b8e1a4 (“Add ESBMC verification harnesses for OpenSMA”, lucascordeiro, late April 2026) as the first integration of ESBMC-based agentic verification harnesses against a production NVIDIA codebase – a timestamp that predates the corresponding arXiv submission [23] by approximately three to four weeks. This publicly verifiable version-control record establishes that our group was the first to implement and deploy an integrated agentic software model-checking architecture on a real-world industrial codebase. In this work, we further reinforce this trajectory: the tight coupling between LLM agents and ESBMC as a formal verification kernel was established progressively from 2023 onwards through automated vulnerability repair [20], LLM-generated loop invariants [21], and the SpecVerify cyber-physical deployment [22], long before the out-of-tree agentic wrapper architecture was publicly formalized [23].

The key structural differentiation between these two parallel tracks lies in the degree of coupling with the underlying solver. The out-of-tree agentic approach [23] treats the formal verifier as a static oracle that accepts or rejects agent-generated candidates, communicating with CBMC and Kani exclusively through their command-line interfaces and witness outputs. The NVIDIA-OpenSMA infrastructure, by contrast, exploits ESBMC’s native multi-solver SMT capabilities and built-in incremental solving passes, as documented in this work. By embedding the neuro-symbolic reasoning loop at the intermediate-representation level – the GOTO program and SSA layer – our implementation retains learned clauses across iterative agent refinements, minimizing the state-space re-exploration overhead that arises when an external agent repeatedly invokes isolated, non-incremental verification queries. This architectural choice consolidates ESBMC as a natively autonomous verification kernel rather than a passive validation backend. It constitutes the primary technical distinction between our prior work and the parallel out-of-tree agentic framework subsequently formalized in the literature [23].

9 Economic and Monetary Impact of ESBMC

We often evaluate formal verification tools in purely technical terms (false-positive rates, benchmark scores, language coverage). Yet, the case for institutional investment ultimately rests on economic reasoning: what losses does the tool prevent, what market does it address, and what commercial activity has it enabled? This section compiles the available evidence for each of these dimensions in ESBMC. Where direct attribution is possible, we present confirmed figures; where it is not, we frame the tool’s impact against authoritative estimates of the cost landscape in which it operates.

9.1 Research Funding and Institutional Investment

The most concrete economic signal for ESBMC is the sustained institutional investment it has attracted. Lucas Cordeiro, ESBMC’s lead developer and director of the SSVLab, reports a career total exceeding USD 20 million in competitive research grants (a self-reported aggregate across multiple currencies at prevailing rates) from agencies including Engineering and Physical Sciences Research Council (EPSRC),² the Royal Academy of Engineering (RAEng),³ the UK Government Communications Headquarters (GCHQ),⁴ the Defence Science and Technology Laboratory (Dstl),⁵ the European Commission,⁶ the Ethereum Foundation,⁷ Intel,⁸ ARM Holdings,⁹ and Brazilian agencies CNPq,¹⁰ FAPESP,¹¹ and CAPES¹² [71]. Table 4 lists seven confirmed grants in which ESBMC is a primary verification component, together representing confirmed public funding of at least £9.3 million and €4.98 million, with two further awards whose individual values the funding bodies have not publicly disclosed.

The confirmed grants in Table 4 account for at least £9.3 million (GBP) and €4.98 million (EUR) in publicly traceable funding from named funders, with two further awards of undisclosed value. The funding bodies report GBP grants at full consortium value; the Soteria award (£5.8M) is a multi-partner program (THG, Manchester, Oxford), and the funding bodies do not separately disclose the Manchester pro-rata share. The >\$20 million career total cited above is a self-reported aggregate across multiple currencies as stated on Cordeiro’s institutional page [71]; the confirmed

²<https://www.ukri.org/councils/epsrc/>

³<https://raeng.org.uk>

⁴<https://www.gchq.gov.uk>

⁵<https://www.gov.uk/government/organisations/defence-science-and-technology-laboratory>

⁶<https://commission.europa.eu>

⁷<https://ethereum.foundation>

⁸<https://www.intel.com>

⁹<https://www.arm.com>

¹⁰<https://www.gov.br/cnpq>

¹¹<https://www.fapeam.am.gov.br>

¹²<https://www.gov.br/capes>

per-grant amounts in Table 4, sourced from UKRI Gateway to Research and EU CORDIS records, should be treated as the authoritative public evidence.

Table 4: Selected public grants in which ESBMC is a primary verification tool. Amounts shown in original grant currency. GBP amounts are EPSRC/UKRI figures from the Gateway to Research database; EUR amounts are from the CORDIS project record

Grant/Programme	Agency	Period	Amount	Notes
EnnCore (EP/T026995/1)	EPSRC	2020–2023	£1 722 000	Neural architecture security; ESBMC verification component
SCorCH (EP/V000497/1)	EPSRC DSbD	2020–2023	£1 036 000	CHERI secure code; produced ESBMC-CHERI [58]
Soteria consortium	UKRI ISCF	2021–2024	£5 800 000	THG, Manchester, Oxford; e-commerce security [72]
H2020 ELEGANT (957286)	EU Commission	2021–2023	€4 983 250	Fully EU-funded; ESBMC as IoT verif. service [73]
Ethereum Consensus (FY22-0751)	Ethereum Found.	2023–2024	N/D	Formal verification of eth2spec Python library [17]
AIcodeRepair	GCHQ	2023–2024	N/D	Automated AI code repair [19]
SECCOM (EP/X037290/1)	Dstl/EPSRC	2023–2025	£789 000	Composable hardware security
Known totals			£9 347 000 + €4 983 250 + 2 undisclosed awards	

Two grants deserve particular attention for their scale and strategic significance. The EU Horizon 2020 ELEGANT project (grant 957286) funded €4 983 250 [73] across twelve consortium partners to deploy ESBMC as an on-demand IoT verification service, representing the first large-scale EU funding of an ESBMC-based industrial platform. The UK Digital Security by Design (DSbD) program, which hosted the SCorCH and Soteria grants, mobilized £70 million in UKRI investment and £270 million in recognized industry co-investment [74]; within this program, ESBMC’s CHERI extension (ESBMC-CHERI) constitutes a direct technical output, providing formal verification support for the capability-based memory safety architecture that DSbD promoted by design.

9.2 Commercial Spin-off: VeriBee

The most direct economic outcome from the ESBMC research program is VeriBee, a software verification startup incorporated in 2025 and founded by Lucas Cordeiro, Richard Allmendinger, and Kaled Alshmrany at the University of Manchester. VeriBee commercializes FuSeBMC – a fuzzing and BMC hybrid engine built on ESBMC internals – as a source-code security product targeting Small and Medium-sized Enterprises (SMEs) [75]. FuSeBMC won the Gold Medal at Test-Comp 2025 and Gold at Test-Comp 2026 and was listed among Ones to Watch in the Tech Climbers Greater Manchester 2025 ranking. The startup has not publicly disclosed any revenue or equity figures. Still, its positioning addresses a market that the Manchester Research Explorer impact report estimates at £3.4 billion annually in losses suffered by UK SMEs from cyberattacks, with a typical incident costing over £6000 per SME [75, 76].

9.3 The Global Software Bug Cost Landscape

ESBMC operates at the intersection of several high-cost problem classes. A 2002 National Institute of Standards and Technology (NIST) study – still the most comprehensive government-commissioned quantification of its kind – estimated that software defects cost the US economy \$59.5 billion annually (approximately 0.6 % of Gross Domestic Product (GDP) at that time), with a feasible reduction of \$22.2 billion achievable through improved testing and verification infrastructure [77].

While this figure predates the modern embedded-systems, cloud, and blockchain economies and we should not treat it as a current estimate, the directional magnitude is consistent with subsequent case studies: a 2020 Consortium for Information and Software Quality (CISQ) report placed the cost of poor software quality in the US alone at \$2.08 trillion [78], and the UK government’s own estimate attributes £27 billion per year in productivity losses to software failures across the economy [79].

Against this backdrop, ESBMC’s positioning – as a formally sound verifier that is also fast enough for CI/CD integration and broad enough to cover C, C++, Rust, Solidity, and Python – addresses a gap that neither lightweight linters nor heavyweight theorem provers fill. The VeriBee startup’s target market of SMEs is particularly apt: SMEs accounts for the majority of software supply-chain touchpoints but has the least capacity to absorb the engineering overhead of traditional formal verification workflows.

IBM’s Systems Sciences Institute documented that fixing a defect after software release costs 60 to 100 times as much as the same fix would cost at the design stage [80]. This ratio is the canonical economic argument for shift-left verification: tools like ESBMC that find bugs during development – rather than after deployment – generate savings that multiply with system scale.

High-profile incidents reinforce this argument. The CrowdStrike outage (July 2024), caused by an out-of-bounds read in the Falcon sensor’s content-configuration file parser, crashed approximately 8.5 million Windows systems and incurred an estimated \$5.4 billion in losses for Fortune 500 companies [81]. This failure falls within the memory-safety class of properties that BMC targets; whether ESBMC could specifically detect this defect would depend on a verification harness and input model not present in any published evaluation.

9.4 DeFi and Blockchain: A High-Stakes Verification Target

Smart contract verification is where the economic argument for ESBMC-Solidity is most directly quantifiable. Blockchain smart contracts are immutable at deployment and directly guard financial assets; a code defect cannot be patched – it can only be exploited. Total Value Locked (TVL) in DeFi protocols reached approximately \$140.7 billion by mid-2025, with Ethereum alone holding roughly \$84 billion at its June 2024 peak [82]. Separately, over \$115 billion in ETH was staked on Ethereum’s Beacon Chain consensus layer as of March 2024 [83].

The scale of historical losses to smart contract exploits establishes the stakes. Halborn’s comprehensive Top 100 DeFi Hacks report (2025) documents \$10.77 billion in total losses across the hundred largest DeFi hacks from 2014 to 2024 [84], while the Chainalysis 2025 Crypto Crime Report records \$2.2 billion stolen across 303 incidents in 2024 alone, a 21 % year-on-year increase [85]. Prominent individual incidents include the Ronin Bridge hack (March 2022, \$625 million, validator logic flaw [86]), the Poly Network exploit (August 2021, \$612 million, smart contract authorisation vulnerability, largely returned), the BNB Bridge attack (October 2022, \$566 million, proof verification failure), the Wormhole Bridge exploit (February 2022, \$320 million, Solidity contract vulnerability), and the original DAO reentrancy attack (2016, \$60 million), which forced Ethereum’s hard fork.

Halborn’s vulnerability classification is directly relevant to ESBMC-Solidity: 34.6 % of on-chain exploits are attributable to input validation and verification failures – precisely the class of property that BMC encodes and verifies [84]. Notably, only 20 % of hacked protocols had received any audit before exploitation, and audited protocols accounted for just 10.8 % of total losses [84], underscoring the protective value of pre-deployment formal verification over post-hoc auditing alone.

ESBMC-Solidity’s published evaluation [3] demonstrates the tool’s practical effectiveness in this context: it detected all vulnerabilities across a benchmark of Solidity contracts drawn from known-vulnerable DeFi protocols – including reentrancy, arithmetic overflow, and array out-of-bounds – with zero false positives on the evaluated set (noting that this benchmark was curated from known-vulnerable contracts, so the result does not generalize to unseen deployment scenarios).

Beyond Solidity, ESBMC-Python [17] identified a previously unreported division-by-zero error in the `integer_squareroot` function of the ECS – the `eth2spec` Python library serving as the normative reference implementation reviewed by dozens of Ethereum Foundation engineers. The finding was confirmed and corrected by specification maintainers. Given that the Beacon Chain consensus layer secured over \$115 billion in staked ETH at the time of discovery [83], and that the Ethereum Foundation raised its maximum bug bounty to \$1 million specifically for consensus-layer vulnerabilities [87], the economic significance of this single automated finding is substantial even without a precise loss-prevention figure.

9.5 Aerospace and Defense: Certification Cost Reduction

DO-178C, the international standard for avionics software certification, imposes one of the most expensive compliance frameworks in software engineering: full certification of a complex avionics system at Design Assurance Level A (DAL A, for failures classified as catastrophic) costs upwards of \$25 million and typically requires five or more years of engineering effort [88]. Developer productivity at DAL A falls to 3–12 source lines of code per day; at typical US aerospace engineering costs, this translates to an effective cost of approximately \$100 per source line [89]. DO-178C’s formal methods supplement (Section 6) explicitly recognizes mathematical proofs as a partial substitute for structural testing, offering a direct cost-reduction pathway for tools such as ESBMC.

The SpecVerify collaboration between the University of Manchester and Lockheed Martin [22] benchmarked the combined LLM + ESBMC pipeline against nine cyber-physical system specifications from Lockheed’s internal benchmark suite, achieving 46.5 % verification accuracy – comparable to NASA’s CoCoSim with fewer false positives. Lockheed Martin invests approximately \$1.5 billion annually in research and development [90]; that a company of this

profile engaged directly with ESBMC on defense-grade specifications provides evidence of perceived industrial value, even in the absence of a published cost-savings figure.

ESBMC’s verification of the ARM Realm Management Monitor (RMM) firmware [91] – a component of ARM’s Confidential Compute Architecture destined for deployment in hundreds of millions of devices – provides another high-leverage data point. The analysis found 23 new property violations that the closest comparable tool (CBMC) missed, including a confirmed critical pointer-to-integer conversion error (see the comparative results table in the experimental evaluation of [91]). ARM Holdings generated \$4.01 billion in revenue in fiscal year 2025 [92]; a firmware vulnerability in ARM Confidential Compute Architecture (CCA) at scale would entail economic exposure far exceeding that of a single certification program.

9.6 Embedded Systems and IoT: The Recurring-Cost Argument

The IBM Cost of a Data Breach 2024 report – based on 604 organizations globally – placed the average breach cost at \$4.88 million (\$10.22 million for US organizations), a 10 % year-on-year increase and the highest figure recorded since the report’s inception [93]. Each compromised record costs an average of \$173. Critically, organizations with AI-assisted security automation reduced their breach costs by \$2.2 million on average relative to those without automation – a figure consistent with the value proposition of ESBMC-AI’s automated repair loop.

IoT device firmware presents a particularly concentrated risk surface: a 2024 Embedded Computing analysis found that the average networked IoT device carries 1267 software components and 1120 Common Vulnerability and Exposures (CVEs), of which 473 are rated Critical or High [94]. National Vulnerability Database (NVD) entries are growing at over 30 000 new CVEs per year, approaching a cumulative total of 250 000. Organizations that outsource CVE remediation have realized savings of \$2.1 million annually compared with in-house patching [95], indicating the magnitude of labor costs that automated verification tooling can displace.

Automotive software is another high-exposure domain: the US National Highway Traffic Safety Administration (NHTSA) recorded 27.7 million vehicle recalls in 2024, with software-related recalls rising from approximately 5 % to 15 % of all recalls by 2023 [96]. Industry analysts project that modern vehicles will contain up to 300 million lines of code within the next decade [96]; ISO 26262 compliance obligations for safety-critical automotive software impose costs analogous to those of DO-178C in aviation.

9.7 Automated Verification vs. Manual Audit: Cost Comparison

ESBMC is distributed as open-source software under a permissive license, rendering its marginal licensing cost zero. Its verification runtime on standard workloads ranges from minutes to hours on commodity hardware. This cost profile compares favorably with the alternatives:

- A professional smart contract audit by a specialist security firm: \$25 000 to \$100 000 for a mid-complexity DeFi protocol [97].
- A commercial formal verification engagement (e.g., Certora, Veridise): typically \$50 000 to \$250 000 per project.
- A penetration test: \$10 000 to \$50 000+ per engagement [98].
- An automated vulnerability scanner (commercial, annual license): \$1000 to \$4500/year.

The ESBMC-AI framework [19] demonstrated that a fully automated pipeline combining ESBMC with an LLM achieves an 80 % success rate in repairing buffer overflow and pointer dereference vulnerabilities across a corpus of 50 000 C programs – without human intervention at any stage. The authors have published no cost-per-fix figure, but the operational cost of the pipeline is bounded by API charges for LLM inference (typically fractions of a cent per small file) plus ESBMC verification time, suggesting per-defect infrastructure costs well under \$1 for simple vulnerabilities. At the scale of the FormAI dataset’s 112 000 programs [63], this represents a verification throughput that no human audit team could match at comparable cost.

9.8 Summary of Economic Evidence

Table 5 consolidates the economic figures discussed in this section, organized into three distinct categories. **Section A** lists confirmed funding directly attributable to ESBMC research; **Section B** presents direct verification outputs produced by ESBMC; **Section C** provides cost-of-failure context – confirmed facts about the world that establish the financial stakes of the problem classes ESBMC addresses, but are not themselves evidence of ESBMC’s impact. Taken together, this evidence shows: ESBMC has attracted confirmed public funding totaling at least £9.3 million and €4.98 million

(see Table 4). This work produced a commercial spin-off (VeriBee), and researchers have applied it to artifacts that guard assets ranging from hundreds of billions of dollars in DeFi capital to safety-critical firmware deployed in hundreds of millions of ARM devices. The economic case for its continued development rests not merely on the cost of running the tool – essentially zero in licensing terms – but on the multiplicative value of finding a defect before deployment rather than after: a ratio that IBM’s historical data places at $60\times$ or more [80].

Table 5: Summary of economic evidence for ESBMC. **Section A** lists confirmed funding attributable to ESBMC. **Section B** lists direct verification results produced by ESBMC. **Section C** provides cost-of-failure context: confirmed facts about the world that establish the financial stakes of the problem classes ESBMC targets; these are *not* direct measures of ESBMC’s impact. All currency values are as reported in the source; GBP/EUR amounts reflect prevailing rates at the time of publication.

Metric	Value	Source/Status
(A) Grant Funding Attributable to ESBMC		
SSVLab career research grants (self-reported aggregate)	>\$20M	Self-reported [71]
EU H2020 ELEGANT grant	€4.98M	Confirmed [73]
Soteria consortium grant (multi-partner; pro-rata not disclosed)	£5.8M	Confirmed [72]
Confirmed public funding (excl. undisclosed awards)	£9.3M + €4.98M	See Table 4
(B) Direct Verification Results (ESBMC outputs)		
ESBMC-AI repair success (buffer overflow/pointer dereference)	up to 80 %	Confirmed [19]
FormAI dataset programs verified	112 000	Confirmed [63]
SpecVerify verification accuracy	46.5%	Proof-of-concept (see Section 8.8) [22]
Ethereum max bug bounty (eth2spec finding)	\$1M	Confirmed [87]
(C) Cost-of-Failure Context (world statistics; not ESBMC-specific)		
US software defect cost (2002)	\$59.5B/yr	Cost-of-failure context [77]
Feasible saving via improved verification	\$22.2B/yr	Cost-of-failure context [77]
UK DSbD prog. (ESBMC-CHERI is one output)	>\$420M	Programme context [74]
DeFi TVL at risk (mid-2025)	\$140.7B	Cost-of-failure context [82]
ETH staked on Beacon Chain	\$115B+	Cost-of-failure context [83]
Top-100 DeFi hacks total losses	\$10.77B	Cost-of-failure context [84]
All crypto stolen in 2024	\$2.2B	Cost-of-failure context [85]
Boeing 737 MAX total exposure	~\$80B	Cost-of-failure context [99]
CrowdStrike outage (Fortune 500)	\$5.4B	Cost-of-failure context [81]
IBM avg. breach cost (2024)	\$4.88M	Cost-of-failure context [93]
AI automation breach saving	\$2.2M/org	Cost-of-failure context [93]
Avg. IoT device CVEs (Critical/High)	473	Cost-of-failure context [94]
UK SME annual cyberattack losses	\$4.28B	Gov. estimate [75]
Typical SME incident cost	\$7560	Gov. estimate [75]
DO-178C DAL-A cost per SLOC	~\$100	Industry estimate [89]
Smart contract audit cost	\$25k–\$100k	Industry [97]

10 Spin-offs, Technology Transfer, and Notable Case Studies

Beyond competition trophies and research publications, ESBMC’s real-world impact is best measured by two complementary dimensions: the derivative projects and institutions it has generated, and the concrete bugs it has found in production-grade software. This section documents both.

10.1 Institutional Spin-offs and Technology Transfer

10.1.1 SSVLAB (UFAM, Brazil)

The most direct institutional spin-off of ESBMC’s founding team is the SSVLab, established by Lucas Cordeiro at the UFAM following the 2009 ASE publication. SSVLab functions as an academic research group whose primary tool is ESBMC. Still, it also serves as a technology-transfer vehicle: the laboratory has collaborated with Brazilian government bodies and industry partners to formally verify embedded and real-time systems, disseminating ESBMC practices within the Brazilian software engineering community. SSVLab is the custodian of the ESBMC GitHub repository and the primary organizing unit behind most language front-end extensions (ESBMC-Solidity, ESBMC-Jimple, ESBMC-CHERI, ESBMC-Python, ESBMC-Rust).

10.1.2 ARM Centre of Excellence in Formal Verification (University of Manchester)

In 2021, Lucas Cordeiro was appointed Director of the ARM Centre of Excellence at the University of Manchester – an industry-academia research center jointly funded by ARM Holdings and the University, focused on applying formal methods to processor architecture, firmware, and system software. ESBMC is the center’s primary model checking asset. This arrangement constitutes a structured technology transfer: ARM gains access to ESBMC capabilities, guided research priorities, and trained postdoctoral researchers; Manchester gains sustained industrial funding, access to proprietary benchmarks, and direct relevance to ARM’s verification pipeline. The ARM Centre collaboration has produced research on verifying CHERI-enabled C programs (ESBMC-CHERI [58]) and on neurosymbolic verification architectures.

10.1.3 ESBMC-AI Open-Source Project

The ESBMC-AI framework (<https://github.com/esbmc/esbmc-ai>) operates as a distinct open-source project layered on top of the core ESBMC engine, maintaining its own release cycle, issue tracker, and contributor community independently of the main ESBMC repository. Structurally it resembles an early-stage product rather than a research prototype: it provides a command-line interface designed for ease of integration into existing developer workflows, documentation aimed at practitioners rather than researchers, and a plug-in architecture that allows the underlying LLM provider – GPT-4, Claude, or open-source models such as LLaMA and Mistral – to be swapped without modifying the verification pipeline itself.

As of 2025, it has attracted external contributors beyond the core ESBMC team, including industry security engineers who have contributed bug fixes, provider integrations, and workflow extensions that reflect real-world deployment requirements rather than purely academic use cases. While no formal commercial entity has yet spun off around ESBMC-AI, the project represents the clearest and most mature candidate for future commercialization within the ESBMC ecosystem: its value proposition – automated, bounded-formally-verified security hardening (violations confirmed within the BMC verification depth) integrated directly into CI/CD pipelines – aligns well with enterprise DevSecOps workflows, where the demand for shift-left security tooling with formal guarantees is growing rapidly across regulated industries.

10.1.4 Rust Foundation Membership

In 2024 ESBMC joined the Rust Foundation’s formal verification initiative [18], gaining formal membership and a seat in the Foundation’s technical working groups dedicated to advancing the safety and correctness guarantees available to Rust developers. This positions ESBMC within the governance structure of the Rust language ecosystem in a manner that goes beyond a mere technical integration: membership carries both privileged research access – to the standard library test suite, the language specification, and pre-publication discussions of language evolution – and a de facto institutional endorsement that strengthens ESBMC’s credibility for commercial Rust verification engagements in competitive procurement contexts.

The collaboration specifically targets the integration of ESBMC as an alternative backend for Kani, the CProver-based Rust model checker maintained by Amazon Web Services, opening a concrete and well-supported pathway to adoption by Rust-using organizations in safety-critical sectors, including automotive, aerospace, and cloud infrastructure, where Rust’s memory safety guarantees are increasingly mandated or strongly preferred by regulators and customers. Should this integration mature, ESBMC would gain access to Kani’s existing user base and validation corpus, substantially accelerating its route to production deployment in the Rust ecosystem.

10.1.5 SpecVerify – University of Manchester/Lockheed Martin Collaboration

The SpecVerify project [22], which combines Claude 3.5 Sonnet with ESBMC for cyber-physical system specification verification, was developed through a direct research partnership with Lockheed Martin, one of the world’s largest defense and aerospace contractors. Lockheed’s active involvement in the project – contributing real system specifications and domain expertise rather than merely lending its name to a publication – indicates genuine institutional interest in evaluating ESBMC-based verification pipelines within defense and aerospace development contexts, given the stringent certification requirements imposed by standards such as DO-178C and MIL-STD-882 in those sectors.

SpecVerify is a documented instance of a defense industrial partner evaluating ESBMC against operational specifications, providing useful evidence of the tool’s applicability beyond academic benchmarks. However, any longer-term deployment or procurement would require substantially more extensive independent validation.

10.2 Notable Case Studies: ESBMC Finding Real Bugs

The following case studies move beyond competition benchmarks to document ESBMC’s effectiveness on production-grade or high-profile artifacts.

10.2.1 ECS – Python

Perhaps the most high-profile bug-finding result in ESBMC’s history came from ESBMC-Python [17], which the authors reported at ISSTA 2024. The ECS – the normative Python reference implementation of the Ethereum proof-of-stake consensus layer – is one of the most widely scrutinized pieces of open-source infrastructure in the blockchain ecosystem, maintained by the Ethereum Foundation and reviewed by dozens of security engineers. ESBMC-Python successfully identified previously unreported defects in this specification. Because the Ethereum consensus layer underpins a network handling hundreds of billions of dollars in assets, the correctness of the specification has significant economic and security implications. The result demonstrated that BMC, when applied through an appropriate language front-end, can detect bugs that slip past intensive manual review and conventional testing.

10.2.2 DeFi Smart Contract Vulnerabilities – Solidity

The ESBMC-Solidity evaluation [3] applied the tool to a benchmark suite of Solidity smart contracts drawn from known vulnerable DeFi protocols. ESBMC-Solidity detected all vulnerabilities in the benchmark set – including arithmetic overflow, reentrancy patterns, and array out-of-bounds accesses – with zero false positives on the evaluated cases. Reentrancy is the vulnerability class responsible for the 2016 DAO hack (approximately \$60M at the time), making its reliable automated detection a commercially significant capability. While the benchmark was curated rather than representing a fresh audit of unknown contracts, the zero-false-positive result at complete detection is notably strong for a formal verification tool on a language with complex execution semantics.

10.2.3 Cyber-Physical Systems at Lockheed Martin – SpecVerify

The SpecVerify evaluation [22] applied the combined LLM + ESBMC pipeline to nine cyber-physical system specifications drawn from realistic defense and aerospace scenarios. The system achieved 46.5 % verification accuracy – defined as the proportion of specifications correctly verified or refuted – while producing fewer false positives than NASA’s CoCoSim, the baseline comparator. While 46.5 % might appear modest, we must contextualize it: formal verification of cyber-physical systems with natural-language-adjacent specifications is an extremely hard problem, and the SpecVerify result represents one of the first demonstrations that a formal verifier can usefully integrate into an industrial Cyber-Physical System specification workflow at all.

Nevertheless, the limitations of this result require explicit acknowledgment. The evaluation corpus of nine systems is small by the standards of empirical software engineering, and its restriction to a single industrial context – defense and aerospace scenarios at Lockheed Martin – limits generalisability to other Cyber-Physical System domains, specification conventions, and safety standards. The accuracy metric itself warrants scrutiny: aggregating correct outcomes across requirements of heterogeneous complexity and criticality obscures the asymmetric cost structure of safety-critical verification, where a missed violation carries categorically greater consequence than a spurious alarm.

The LLM component additionally introduces non-determinism that may affect reproducibility across model versions, prompt formulations, and deployment environments, a concern that the current evaluation does not systematically address. Reaching production-grade performance would require, at minimum: a substantially larger and more diverse benchmark suite spanning recognized safety standards such as DO-178C, IEC 61508, and ISO 26262; per-category accuracy reporting disaggregated by requirement type and criticality level; human-in-the-loop validation of LLM-generated specifications before formal checking; and rigorous ablation studies isolating the contribution of each pipeline component. Until we have such evidence, we should best understand the 46.5 % result as an initial feasibility demonstration – evidence that the LLM+formal-verifier architecture is viable – rather than a deployment-ready performance baseline.

10.2.4 Self-Healing Software Demonstration – Security Defects in C

The self-healing software paper [19], presented at AST 2025, constructed a controlled but realistic experiment: a corpus of C programs with deliberately introduced security-relevant defects (buffer overflows, null pointer dereferences, integer overflows, format string vulnerabilities). ESBMC first detected the defects formally, then the LLM repair loop proposed patches, and ESBMC re-verified the results. The combined pipeline successfully repaired up to 80 % of buffer-overflow and pointer-dereference defects without human intervention and, critically, without introducing new formal violations detectable by ESBMC. The repair success rate on buffer overflows and pointer defects is particularly notable because

these are the vulnerability classes most commonly exploited in memory-unsafe C code. The study provides empirical support for the claim that ESBMC-AI is not merely a research curiosity but a plausible automated security-hardening tool.

10.2.5 LLM-Generated Code Security Assessment – FormAI Dataset

The FormAI dataset study [63] used ESBMC as a large-scale oracle to formally assess the security properties of code generated by seven different LLMs across 112 000 programs. The key finding was that the majority of LLM-generated C programs contained at least one formally verifiable safety violation, with significant variation across model families and prompt strategies. This large-scale bug-finding exercise constituted a direct use of ESBMC as a bug-finding tool at an industrial scale: running formal verification across a dataset orders of magnitude larger than typical research benchmarks. The study provided one of the first formally grounded, large-scale characterizations of LLM code-generation quality, influencing subsequent work on code-generation model evaluation and reinforcement learning from formal feedback.

10.2.6 Embedded and Firmware Verification – Industrial Collaborations

ESBMC’s original design target – embedded ANSI-C software – has generated a sustained stream of industrial engagements that are less publicly documented than the above, partly for confidentiality reasons. Academic-industrial collaborations facilitated by SSVLab and the ARM Centre of Excellence have used the tool to verify firmware for automotive microcontrollers, IoT device drivers, and safety-critical controllers. Common bug classes include arithmetic overflow in fixed-point signal-processing routines, buffer overruns in packet parsers, and data races in interrupt-handler/main-loop shared-memory patterns. These engagements inform ESBMC’s operational models, library coverage, and performance-tuning priorities – the bug classes that recur in industrial code drive the verifier’s development roadmap more directly than any competition category.

11 Challenges and Future Trends

11.1 Scalability to Large Codebases

Scalability to large, real-world codebases remains a fundamental challenge due to the state-space explosion problem [43]: formula size grows with program complexity, loop bounds, and thread count. Several mitigation strategies are already deployed in ESBMC or are natural near-term extensions. **Incremental SMT solving** (Section 5.3), available via `-incremental-bmc`, converts the quadratic re-solve cost of Naïve depth-stepping into a linear one [9], and is the first line of defense for programs requiring deep unrolling. **Compositional verification** decomposes a program into modules verified independently with assume-guarantee contracts [55], reducing each monolithic formula to smaller, tractable subproblems. **Abstract interpretation** frameworks (e.g., Inference Kernel for Open Static Analyzers (IKOS), Facebook Infer) supply pre-verified loop summaries to the BMC engine, reducing unrolling depth. **Portfolio-solving** strategies predict which solver will handle a formula fastest, leveraging the competitive SMT landscape [14, 15]. LLMs can also suggest program decompositions and contract candidates [69], lowering the manual effort required for compositional verification at an industrial scale.

11.2 Neurosymbolic Verification

ESBMC-AI instantiates a **broader neurosymbolic vision**: combining neural pattern recognition with the formal guarantees of symbolic reasoning. The paradigm promises verification workflows that are more accessible to non-expert developers and more trustworthy than purely LLM-driven approaches. Realizing this promise requires resolving several fundamental open problems.

The most urgent is **reproducibility under LLM non-determinism**. The same prompt, at the same temperature, can produce a correct repair on one run and a subtly incorrect one on another [68]. In safety-critical contexts, where reproducibility is a regulatory requirement, this non-determinism is a structural obstacle. Current mitigations – multiple-sample aggregation and formal filtering layers that reject proposals failing ESBMC’s harness [68] – add latency without fully resolving the problem. The core research question is: which architectural and training-time interventions make LLM outputs reproducible with a specified confidence level, and how should that level relate to the system’s safety integrity level during verification?

The second open problem is **completeness in LLM-generated specifications**. Translating informal requirements into formal assertions that ESBMC can check remains largely unsolved [100]. Current models produce syntactically plausible assertions, but their completeness – capturing all intended violations, not just obvious ones – is hard to guarantee. An incomplete specification provides false assurance: the verification result is technically valid while

substantively incomplete. Progress is likely to require collaboration between the natural language processing and formal methods communities, drawing on requirement taxonomies from DO-178C and ISO 26262.

The third open problem is **fine-tuning on formal verification corpora**. The dominant paradigm remains prompting-only: we expect a general-purpose model to generalize from its pretraining distribution [22]. Datasets such as FormAI [63] – pairing C programs with formal vulnerability annotations – provide a foundation for supervised fine-tuning on verification-specific tasks. Key questions are whether fine-tuned models generalize across languages and formalisms, whether fine-tuning improves worst-case reliability, and whether their gains complement retrieval-augmented generation over verified code corpora.

A fourth open problem is **the explanation and accessibility of counterexamples** [101]. ESBMC produces precise but terse traces that require expertise in formal methods to interpret; this gap is a significant barrier to adoption. LLM-mediated translation into natural language is promising, but faithfulness is the key challenge: a fluent but inaccurate explanation may misdirect remediation, introducing new defects. Structured output formats and retrieval-augmented pipelines that anchor LLM outputs to specific trace elements are promising directions, but the systematic evaluation of their faithfulness remains an open question.

11.3 Expanding Language Coverage

While ESBMC currently supports nine language front-ends (ANSI-C, C++03, CUDA, Solidity, Kotlin, Cheri C, C++11+, Python, and Rust – where C++11+ is counted as a distinct modernised front-end relative to C++03, though both target the same language family; treating them as one gives eight distinct languages), **high-value gaps remain** across several target-language categories, each with its own verification motivation and integration challenges.

The most immediate gap is **mainstream general-purpose languages**. JavaScript/TypeScript and Java dominate enterprise and web development, yet their absence leaves Android applications, server-side microservices, and front-end logic inaccessible to ESBMC-based verification. The ESBMC-Jimple frontend – which targets Kotlin via the Jimple IR, the standard representation for Java bytecode in the Soot framework – provides a template, making full Java coverage a near-term, tractable objective.

WebAssembly is an increasingly urgent verification target. Originally a portable web compilation target, it now runs in embedded firmware, on smart contract platforms (as an alternative to the EVM), in serverless functions, and in plugin sandboxes. Its sandboxed memory model and typed instruction set favor formal analysis. Since C, C++, and Rust – all supported by ESBMC – compile to WebAssembly, a frontend could reuse existing verification machinery. Verifying at the WebAssembly level is especially valuable when the compilation toolchain is untrusted or when deployment-time optimizations introduce behaviors absent from the source.

Hardware description languages are another critical gap. ESBMC-Cheri showed that ESBMC can encode hardware architectural constraints as SMT verification conditions. This methodology extends naturally to VHSIC Hardware Description Language (VHDL) and SystemVerilog, the dominant languages for Field-Programmable Gate Array (FPGA) and Application-Specific Integrated Circuit (ASIC) design, enabling co-verification with the software layers that drive them. Such hardware-software co-verification is increasingly demanded, with Design Assurance for Airborne Electronic Hardware (DO-254) governing airborne electronic hardware and DO-178C governing software.

Formal annotation languages such as JML and ACSL also fall outside ESBMC’s current scope. Both annotate Java and C programs with pre- and post-conditions and invariants. Integrating ESBMC as a backend checker for these annotations would position it within established workflows for high-assurance avionics and medical device software.

The most structurally novel gap is **polyglot cross-language contract verification**. Modern safety-critical architectures combine a Python orchestration layer, a Rust systems library, and a C firmware component – each verified in isolation. No current ESBMC mechanism checks interface contracts across languages, yet a property violation may only manifest when components are composed. This verification gap requires a unified semantic model that covers multiple type systems, memory models, and concurrency semantics, along with a compositional strategy for assigning blame to the correct component. The core research question is: which intermediate representation would enable ESBMC to verify cross-language contracts and handle semantic mismatches – such as Python’s dynamic typing meeting Rust’s ownership model – at language boundaries?

11.4 AI Agent Architectures

Research is advancing toward **autonomous verification agents** that orchestrate static analysis, fuzzing, symbolic execution, and formal verification within a workflow guided by an AI planner. ESBMC’s programmatic interface, broad language coverage, and rich output – counterexamples, proofs, and coverage reports – position it well as a formal

verification oracle. This alignment with the emerging autonomous-agent paradigm reflects a broader shift: rather than treating formal verification as a standalone expert-operated tool, the emerging paradigm embeds verifiers as callable, trustworthy components within larger autonomous pipelines.

The typical architecture involves an **LLM-based planner** that decomposes a verification goal into subtasks, selects tools, interprets outputs, and iterates until it finds a proof, finds a counterexample, or exhausts resources. ESBMC occupies a privileged role in this template: unlike static analyzers or fuzzers, its outputs carry formal semantics – a counterexample witnesses a genuine reachable violation, and a successful run provides a bounded correctness guarantee. This precision makes ESBMC a trustworthy oracle rather than a heuristic signal source.

Concrete instantiations have already emerged. The ESBMC-LLM framework [19] demonstrated that an LLM can consume ESBMC counterexamples and generate targeted repairs, closing the loop between verification failure and automated remediation. SpecVerify [22] inserts an LLM-based natural language understanding stage upstream of formal checking. Together, these illustrate a convergence: ESBMC is evolving from an expert-operated tool into a verification service that autonomous agents invoke programmatically.

Researchers must address several challenges before these architectures reach production maturity. Planners must reason reliably about the scope and soundness of ESBMC’s guarantees, distinguishing when bounded results are sufficient from when unbounded reasoning is required. We must manage LLM-based non-determinism to ensure reproducible pipeline outputs. Tool interfaces must be standardized so that agents can invoke ESBMC alongside AFL++, Infer, and KLEE without bespoke integration. Trust calibration is also open: over-reliance on bounded guarantees risks missing properties that require stronger methods, while under-reliance wastes resources. These challenges demand new theoretical frameworks for composing heterogeneous verification tools within an autonomous planner.

11.5 Compliance and Certification

Safety standards – DO-178C, ISO 26262, and IEC 61508 – increasingly recognize **formal verification as a compliance means** [62, 1, 2]. ESBMC’s CFG-based branch coverage instrumentation [54] directly targets the structural coverage criteria (MCDC, decision coverage) these standards demand. Substantial work remains across five areas.

The first is **traceable proof artifacts**. We must link machine-readable correctness witnesses [14] to specific requirements in auditor-acceptable form. Currently, bridging a formal verification result to a certification evidence package requires significant expert effort. Closing this gap requires a shared ontology: a formally defined mapping between ESBMC’s artifacts – counterexample traces, proof certificates, coverage reports – and the evidence categories recognized by each standard.

The second is **assurance-case construction**. Auditors consume structured safety cases, not raw verification results. The dominant notations are the Goal Structuring Notation (GSN) and Architecture Analysis and Design Language (AADL) safety annexes, both machine-readable and, in principle, auto-populatable from verification evidence. Linking ESBMC’s witnesses to GSN goal nodes or AADL elements would automatically flow results into auditor-facing structures, maintaining a live traceability chain. This approach is technically feasible with tooling such as ACE and Advocate, but requires ESBMC to emit structured, schema-conformant artifacts.

The third is **programmable logic certification under DO-254**. ESBMC-CHERI demonstrated that ESBMC can encode hardware architectural constraints as SMT verification conditions. DO-254 governs FPGA and ASIC development in avionics, demanding design assurance comparable to DO-178C for software, yet formal verification tooling for hardware description languages remains immature. Extending ESBMC to VHDL and SystemVerilog would address a significant unmet need and create a natural bridge to hardware-software co-verification workflows.

The fourth is **operational models for safety-critical libraries**. Developing and validating models for Aeronautical Radio, Incorporated (ARINC) APEX, AUTomotive Open System ARchitecture (AUTOSAR) RTE, and POSIX real-time extensions – enabling ESBMC to verify application code without access to library source code – remain significant barriers. Models must be faithful enough to produce meaningful results yet abstract enough to remain SMT-tractable. Validating them against real implementations is itself a verification problem, and developing and maintaining them across platforms is a recurring cost the community has not found a scalable way to absorb.

The fifth and most significant is **tool qualification under Software Tool Qualification Considerations (DO-330)**, the tool qualification supplement to DO-178C. For ESBMC to serve as a verification tool in a DO-178C-certified program – rather than a development aid whose outputs require independent checking – we must qualify it at a level commensurate with the system’s design assurance level. DO-330 qualification requires: a tool operational requirements document specifying ESBMC’s intended behavior and coverage boundaries; a tool verification plan; anomaly reporting and resolution processes; and a configuration management system that ensures developers use the qualified version in certified development.

For an actively developed research tool, **the tension between rapid feature evolution and DO-330’s configuration management discipline is acute**: each capability-changing release may require re-qualification. A pragmatic path mirrors the approach taken by AbsInt for ASTRÉE and AdaCore for GNAT Ada compiler (GNAT) Pro – qualifying a stable, feature-frozen ESBMC configuration while continuing research development on an unqualified branch. Establishing this pathway requires sustained engagement with certification authorities and potentially a commercial entity – such as an expanded VeriBee – capable of maintaining a qualified configuration over a product lifetime measured in decades.

11.6 Quantum Software and Hardware-Software Co-Verification

ESBMC’s SMT-based foundations are theoretically applicable to quantum circuit verification as quantum computing matures. ESBMC-CHERI has already demonstrated the tool’s capacity to reason about hardware-software interfaces, positioning it to support the integration of hardware abstraction models for heterogeneous System on Chip (SoC), FPGA, and accelerator verification. Both directions share a common challenge: they require reasoning beyond the classical sequential semantics for which BMC was designed, demanding principled extensions to ESBMC’s core theory.

Quantum software verification is becoming practically urgent. IBM, Google, and IonQ make their platforms cloud-accessible, and developers write quantum algorithms in Qiskit, Cirq, and Q#. These programs are not immune to defects: incorrect qubit initialization, erroneous gate sequences, and entanglement errors produce silently incorrect results that testing alone cannot catch, as quantum measurement is probabilistic and destructive. SMT theories for complex-valued linear algebra and tensor products are an active research area [102], and tools such as QMC and Quartz demonstrate that bounded model checking transfers to quantum circuits [103, 104]. ESBMC’s solver-agnostic architecture, already supporting floating-point and bit-vector theories via pluggable backends, provides a natural integration point. A near-term trajectory could extend ESBMC with a quantum-circuit frontend (OpenQASM or Qiskit IR) and a quantum-state-theory plugin, enabling bounded verification of unitarity, entanglement invariants, and measurement-outcome distributions.

Hardware-software co-verification is an equally compelling extension. Modern SoCs combine Central Processing Unit (CPU) cores, GPU accelerators, FPGA fabrics, and cryptographic coprocessors, each with distinct semantics and security boundaries. ESBMC-CHERI [58] demonstrated feasibility by encoding CHERI’s capability-based memory protection model as SMT constraints, enabling reasoning about memory safety properties that depend on hardware-enforced invariants. The methodology extends to VHDL and SystemVerilog designs co-verified with their software drivers, and to accelerator models such as CUDA – already supported by ESBMC-GPU [105] – and emerging standards such as SYCL and oneAPI, with hardware fidelity models capturing memory hierarchy behavior and warp divergence.

Realizing either direction at production quality requires **sustained investment in three areas**. First, the ESBMC team must develop and integrate new SMT theories in collaboration with the SMT community. Second, frontend toolchains must translate quantum circuit representations and hardware description languages into ESBMC’s verification harness – an effort comparable to ESBMC-Solidity or ESBMC-CHERI. Third, the community must establish benchmark suites and ground-truth corpora for empirical evaluation, which is a prerequisite for community validation through competitions such as SV-COMP.

11.7 Real-Time and Timing Constraint Verification at Scale

Correct timing is a first-class safety requirement in embedded and avionics domains, yet it remains one of ESBMC’s least developed verification dimensions. ESBMC has demonstrated bounded model checking of timing constraints [56], establishing theoretical feasibility. The gap between this proof-of-concept and the demands of a DO-178C or DO-254 certified program, however, is substantial.

Worst-Case Execution Time (WCET) analysis asks: what is the longest possible execution time over all inputs and paths? WCET bounds are required inputs to schedulability analysis, and overly conservative estimates waste processor capacity and increase system cost. Current tools such as AbsInt’s Timing Analyzer (AIT) and Rapita’s RVT combine static analysis with hardware timing models but operate independently of functional verifiers. Integrating WCET reasoning into ESBMC’s BMC framework – encoding timing models as SMT constraints alongside functional properties – would enable simultaneous verification of correctness and timing, a combination no current tool provides.

The challenge compounds for concurrent Real-Time Operating System (RTOS) tasks. A single task’s WCET does not determine system-level timing correctness; that requires reasoning about scheduling, preemption, resource contention, and priority inversion. ESBMC’s existing concurrency machinery provides a partial foundation, but encoding the scheduler’s semantics within the verification model is a non-trivial scalability challenge. The key research

question is: which SMT encoding of RTOS scheduling and hardware timing models would allow ESBMC to verify composed timing constraints with sufficient fidelity for DO-178C scheduling analysis? Addressing this would position ESBMC as a unified functional and temporal verification platform – a capability no current open-source tool provides.

11.8 Counterexample Intelligibility and Human Factors

The economic argument for ESBMC depends on **the tool’s ability to change developer behavior**. A counterexample that a developer cannot interpret within a reasonable time budget does not change behavior, regardless of its formal correctness. ESBMC can produce traces spanning hundreds of SMT assignment steps. For a formal methods expert, such a trace is informative. For a software engineer without formal methods training – the population ESBMC must reach for industrial adoption – it is effectively opaque.

This gap is a primary barrier to industrial adoption. Surveys consistently identify counterexample interpretation as the dominant friction point [106, 107], ahead of installation, performance, and licensing. The problem has three dimensions. Length and navigability: a 200-step trace presents a navigation problem before an interpretation problem; engineers need structured visualization to identify causally relevant steps without reading sequentially. Abstraction level: SMT-level traces expose bit-vector assignments and solver-introduced variables that obscure the source-level story. Domain relevance: an engineer on an AUTOSAR component needs explanations in terms of runnables and ports, not raw memory operations.

Addressing these dimensions requires **both technical and empirical work**. Structured trace visualization tools – analogous to interactive debuggers but driven by formal semantics – could let engineers navigate traces at multiple abstraction levels, collapsing irrelevant steps. Domain-specific explanation templates could translate SMT assignments into vocabulary familiar to the target domain. LLM-mediated explanation is promising, but faithfulness must be a hard constraint: a fluent but inaccurate explanation misdirects remediation. On the empirical side, the community needs agreed usability metrics – time-to-correct-interpretation, root-cause error rate, and fix success rate – to evaluate and compare approaches.

11.9 Polyglot and Cross-Language System Verification

ESBMC supports nine languages across embedded, systems, smart contract, and scientific computing domains, but **the current architecture treats each frontend as an independent verification pathway**. It verifies a Python program as a Python program and a Rust program as a Rust program – the two never interact. This independence is increasingly at odds with the requirements of modern safety-critical systems.

Contemporary architectures combine a Python orchestration layer, a Rust systems library, and a C firmware component – each communicating through well-defined APIs. Each component may be individually correct, yet **the composition can violate a system-level safety property** if interface contracts are misspecified or silently violated. No current ESBMC mechanism addresses cross-language compositional verification, and the gap is not merely a frontend engineering problem.

Verifying cross-language contracts requires solving three interrelated challenges. First, a unified semantic model must represent multiple type systems, memory models, and concurrency semantics without collapsing verification-relevant distinctions: Python’s dynamic typing, Rust’s ownership model, and C’s manual memory management make fundamentally different safety guarantees. Second, a compositional reasoning strategy must assign blame for violations to the correct component, enabling actionable counterexamples. Third, we must integrate interface specification languages – drawing on session types or contract-based design frameworks – into ESBMC’s harness. The core question is: which intermediate representation and compositional framework would enable sound verification of cross-language contracts, including semantic mismatches at Foreign Function Interface (FFI) boundaries?

11.10 Reproducibility and LLM Non-Determinism in Hybrid Verification Pipelines

The reproducibility problem is a structural property of any hybrid verification pipeline incorporating LLM components – not specific to any particular ESBMC deployment. LLM outputs are non-deterministic: the same prompt, same model, and same configuration can produce different outputs due to temperature sampling, batching decisions, and silent model updates. This non-determinism poses a fundamental obstacle to industrial certification of hybrid pipelines.

Certification frameworks – DO-178C, ISO 26262, IEC 61508 – assume that **a verification tool produces the same outputs given the same inputs**. A correctness certificate is meaningful only if the tool can regenerate it on demand. A hybrid pipeline in which an LLM may produce a different repair, invariant, or specification on each invocation cannot satisfy this requirement without architectural constraints that the current ESBMC-AI framework does not impose.

The problem manifests in three forms. Run-to-run non-determinism – variation across invocations of the same model with the same prompt – can be mitigated by setting the temperature to 0 and fixing the random seed, though infrastructure-level variation may persist. Version-to-version non-determinism – introduced by model updates – requires pinning to a specific version or maintaining a version-tagged corpus of results to detect regressions. Prompt-sensitivity non-determinism – variation from minor prompt reformulations that preserve intent – is the least tractable, reflecting genuine uncertainty in the model’s generalization behavior. The core research question is what architectural constraints and formal filtering layers make a hybrid LLM+ESBMC pipeline reproducible to a specified confidence level, and how that level should relate to the system’s safety integrity level.

11.11 Open-Source Sustainability and Governance

ESBMC’s **open-source model** has enabled distributed contributions from Manchester, UFAM, and an international community, facilitated adoption in benchmarking competitions, and provided the transparency required for independent validation. As ESBMC transitions toward industrial deployment – with VeriBee at the forefront of commercialization – governance and sustainability challenges warrant explicit acknowledgment.

The most immediate tension is **potential divergence between the open-source research branch and a commercial VeriBee branch**. Commercial deployment demands configuration stability and regression testing that a research cadence struggles to sustain. If the commercial branch incorporates proprietary extensions that developers do not contribute back, the result is fragmentation: academic contributors improve the branch, while industrial users do not deploy it, and vice versa. Managing this requires explicit governance decisions about branching strategy, contribution policy, and upstreaming conditions – decisions the ESBMC community has not yet had to make explicitly.

Contributor sustainability is a related concern. ESBMC’s development has relied on PhD students and postdocs whose project funding bounds their tenure. Institutional knowledge of architectural decisions and solver subtleties is not fully documented, which represents a concentration risk. As scope expands to nine languages and multiple AI-augmented pipelines, the maintenance surface grows faster than the contributor base – a technical debt risk common in research software but rarely acknowledged in capability-focused surveys.

API stability is a third challenge. The ESBMC-AI framework, the Kani backend, and future CI/CD integrations depend on the stability of ESBMC’s programmatic interface across releases. Research tools typically make no stability guarantees, and breaking changes impose significant downstream maintenance costs. Establishing a versioned API with documented stability – analogous to LLVM or Z3 – would reduce integration friction but would require governance commitment and release engineering discipline that academic projects rarely sustain without dedicated funding. Addressing these sustainability challenges is not peripheral; it is a precondition for the sustained industrial adoption that the economic argument presupposes.

12 Conclusion

ESBMC has undergone substantial and sustained development over fifteen years: from a research prototype for verifying embedded ANSI-C programs using SMT solvers [8] to a versatile, industrially relevant, AI-integrated formal verification platform with confirmed deployments spanning embedded firmware, blockchain smart contracts, defense and aerospace cyber-physical systems, and autonomous software repair [19, 22, 3, 17]. Its core innovation – the native use of SMT solving for program verification, as formalised in the extended IEEE Transactions on Software Engineering (TSE) paper [8] – has proven durable and generative, enabling principled extensions to floating-point arithmetic [57], concurrency [55, 12], and a nine-language verification portfolio covering ANSI-C, C++03, CUDA, Solidity, Kotlin, CHERI C, C++11+, Python, and Rust [16, 3, 17, 18].

Five themes emerge from this survey as defining characteristics of ESBMC’s trajectory:

1. **Theory-driven engineering:** Each major capability is grounded in well-understood theoretical results – k -induction [44, 45], the DPLL(T) solver architecture [46], floating-point SMT theory [57], and context-bounded concurrency [55] – providing a sound basis for extension and composability that has sustained relevance across three generations of hardware and language targets.
2. **Competition as a quality driver:** Systematic participation in SV-COMP [13, 14] and Test-Comp, accumulating 43 awards (35 SV-COMP medals + 8 Test-Comp medals) under rigorous independent evaluation [13, 14, 15], has provided objective external feedback that has motivated successive algorithmic improvements – from incremental SMT solving and interval analysis [11] to the concurrent scheduling advances of v7.7 [12, 54] – reflected in each successive release.

3. **Language pluralism:** Rather than treating C as the definitive target, the ESBMC team has systematically extended the tool to nine languages by 2025, enabled by the common IR architecture that decouples front-end language coverage from back-end verification power [9], and cemented by institutional partnerships including Rust Foundation membership [18] and the Lockheed Martin collaboration [22].
4. **AI integration without abandoning rigour:** The ESBMC-AI programme [19, 21] integrates LLMs as hypothesis generators, repair proposers, loop invariant suppliers, and specification translators while retaining formal verification as the sole arbiter of correctness – achieving up to 80 % automated repair success on buffer-overflow and pointer-dereference categories, on security-critical C programs [19] and avoiding the pitfalls of AI-only approaches [68, 67].
5. **Demonstrated economic and societal value:** Beyond technical metrics, ESBMC has generated measurable real-world impact: confirmed public funding of at least £9.3 million and €4.98 million across seven grants [71, 73, 72], the VeriBee commercial spin-off [75], confirmed bug findings in the Ethereum Consensus Specification [17] and DeFi smart contracts [3], and a defence industrial deployment at Lockheed Martin [22] – establishing a precedent for university-developed formal verification tools achieving production-grade industrial adoption.

A structured research agenda shapes the near-term future for ESBMC: resolving LLM non-determinism and reproducibility in hybrid pipelines [68], improving counterexample intelligibility to reduce the interpretive burden on non-expert engineers [106, 107], enabling polyglot cross-language contract verification across heterogeneous system stacks, advancing real-time and WCET constraint analysis for avionics and automotive targets [56], pursuing DO-330 tool qualification and safety certification support for DO-178C and ISO 26262 [62, 54], and specialising LLM components through fine-tuning on formal verification corpora such as FormAI [63]. The long-term trajectory – quantum software verification, universal hardware-software co-verification across heterogeneous SoCs and FPGAs, and certification-grade formal assurance at industrial scale [4] – points to a tool that has established the institutional, technical, and commercial foundations to remain at the frontier of formal verification for decades to come.

Acknowledgments

The authors would like to express their gratitude to the Department of Computer Science at the University of Manchester (UoM) and the Systems and Software Security (S3) Research Group for their invaluable support, collaborative environment, and access to cutting-edge resources, which were instrumental in the success of this research. We conducted this work with partial funding from the Engineering and Physical Sciences Research Council (EPSRC) grants EP/T026995/1, EP/V000497/1, EP/X037290/1, and the Soteria project, awarded by the UK Research and Innovation under the Digital Security by Design (DSbD) Programme.

Conflict of Interest

Lucas Cordeiro is the founding developer and lead researcher of ESBMC and a co-founder of VeriBee Ltd. This commercial spin-off commercializes technology derived from the ESBMC research program. Pierre Dantas and Waldir Junior actively contribute to the ESBMC research program at their respective institutions. These relationships constitute direct conflicts of interest that readers should weigh when evaluating claims regarding ESBMC’s capabilities, competitive standing, and economic impact. Members of the ESBMC team co-author approximately 23 % of references in this survey; as explained in Section 1.1, this proportion is structurally unavoidable for a single-tool survey. The authors declare that no external funding specifically supported the preparation of this survey article, and that primary sources identified in the text provide all quantitative claims.

References

- [1] John Hatcliff, Andrew King, Insup Lee, Alasdair Macdonald, Anura Fernando, Michael Robkin, Eugene Vasserman, Sandy Weininger, and Julian M. Goldman. Rationale and Architecture Principles for Medical Application Platforms. In *2012 IEEE/ACM Third International Conference on Cyber-Physical Systems*, page 3–12, Beijing, China, April 2012. IEEE. doi: 10.1109/iccps.2012.9.
- [2] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon, and Michael Fisher. Formal Specification and Verification of Autonomous Robotic Systems: A Survey. *ACM Computing Surveys*, 52(5):1–41, September 2019. ISSN 1557-7341. doi: 10.1145/3342355.
- [3] Kunjian Song, Nedas Matulevicius, Eddie B. de Lima Filho, and Lucas C. Cordeiro. ESBMC-solidity: an smt-based model checker for solidity smart contracts. In *Proceedings of the ACM/IEEE 44th International*

- Conference on Software Engineering: Companion Proceedings*, ICSE '22, page 65–69, Pittsburgh, PA, USA, May 2022. ACM. doi: 10.1145/3510454.3516855.
- [4] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):1–36, October 2009. ISSN 1557-7341. doi: 10.1145/1592434.1592436.
 - [5] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of Model Checking*. Springer International Publishing, Cham, Switzerland, 2018. ISBN 9783319105758. doi: 10.1007/978-3-319-10575-8.
 - [6] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic Model Checking without BDDs*, page 193–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 9783540490593. doi: 10.1007/3-540-49059-0_14.
 - [7] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, July 2001. ISSN 1572-8102. doi: 10.1023/a:1011276507260.
 - [8] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. *IEEE Transactions on Software Engineering*, 38(4):957–974, July 2012. ISSN 0098-5589. doi: 10.1109/tse.2011.59.
 - [9] Mikhail R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. ESBMC 5.0: an Industrial-Strength C Model Checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE'18, page 888–891, Manchester, UK, September 2018. ACM. doi: 10.1145/3238147.3240481.
 - [10] Kunjian Song, Mikhail R. Gadelha, Franz Brauße, Rafael S. Menezes, and Lucas C. Cordeiro. *ESBMC v7.3: Model Checking C++ Programs Using Clang AST*, page 141–152. Springer Nature Switzerland, Cham, Switzerland, December 2023. ISBN 9783031493423. doi: 10.1007/978-3-031-49342-3_9.
 - [11] Rafael Sá Menezes, Mohannad Aldughaim, Farias, et al. *ESBMC v7.4: Harnessing the Power of Intervals: (Competition Contribution)*, page 376–380. Springer Nature Switzerland, Luxembourg City, Luxembourg, 2024. ISBN 9783031572562. doi: 10.1007/978-3-031-57256-2_24.
 - [12] Rafael Menezes et al. ESBMC v7.7: Efficient Concurrent Software Verification with Scheduling, Incremental SMT and Partial Order Reduction. In *Proceedings of the 31st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2025)*, Lecture Notes in Computer Science, Hamilton, New Zealand, 2025. Springer. doi: 10.1007/978-3-031-90660-2_16.
 - [13] Dirk Beyer. Competition on Software Verification (SV-COMP). In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012)*, volume 7214 of *Lecture Notes in Computer Science*, pages 504–524, Tallinn, Estonia, 2012. Springer. doi: 10.1007/978-3-642-28756-5_38.
 - [14] Dirk Beyer. *State of the Art in Software Verification and Witness Validation: SV-COMP 2024*, page 299–329. Springer Nature Switzerland, Luxembourg City, Luxembourg, 2024. ISBN 9783031572562. doi: 10.1007/978-3-031-57256-2_15.
 - [15] SSVLab. ESBMC: An Industrial-Strength C Model Checker - Competition Results. <https://ssvlab.github.io/esbmc/sv-comp.html>, 2024. Accessed: May 2026.
 - [16] Felipe R. Monteiro, Mikhail R. Gadelha, and Lucas C. Cordeiro. Model Checking C++ Programs. *Software Testing, Verification and Reliability*, 32(1):e1793, September 2021. ISSN 1099-1689. doi: 10.1002/stvr.1793.
 - [17] Felipe R. Farias, Rafael Menezes, and Lucas C. Cordeiro. ESBMC-Python: A Bounded Model Checker for Python Programs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*, Vienna, Austria, 2024. ACM. doi: 10.1145/3650212.3685304.
 - [18] Rust Foundation. Expanding the Rust Formal Verification Ecosystem: Welcoming ESBMC. <https://rustfoundation.org/media/expanding-the-rust-formal-verification-ecosystem-welcoming-esbmc/>, 2024. Accessed: May 2026.
 - [19] Norbert Tihanyi, Yiannis Charalambous, Ridhi Jain, Mohamed Amine Ferrag, and Lucas C. Cordeiro. A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification. In *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 136–147, Ottawa, ON, Canada, apr 28 2025. IEEE, IEEE. doi: 10.1109/ast66626.2025.00020.

- [20] Yiannis Charalambous, Edoardo Manino, and Lucas C. Cordeiro. Automated Repair of AI Code with Large Language Models and Formal Verification, 2024.
- [21] Muhammad A. A. Pirzada, Giles Reger, Ahmed Bhayat, and Lucas 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*, pages 1395–1407, Sacramento, CA, USA, oct 27 2024. ACM, ACM. doi: 10.1145/3691620.3695512.
- [22] Weiqi Wang, Marie Farrell, Lucas C. Cordeiro, and Liping Zhao. Supporting software formal verification with large language models: An experimental study. In *2025 IEEE 33rd International Requirements Engineering Conference (RE)*, page 423–431, Reykjavik, Iceland, September 2025. IEEE. doi: 10.1109/re63999.2025.00049.
- [23] Youcheng Sun, Jiawen Liu, Daniel Kroening, and Jason Xue. Agentic model checking, 2026.
- [24] Alan Cheng and Lucas C. Cordeiro. NVIDIA-OpenSMA: ESBMC verification harnesses for OpenSMA (vr_sma branch). https://github.com/lucasccordeiro/NVIDIAOpenSMA/commits/vr_sma/, 2025. Repository initialised 30 July 2025 (commit 0dcdcd1); first verification commit 9b8e1a4 (“Add ESBMC verification harnesses for OpenSMA”), April 2026.
- [25] Barbara Kitchenham and Stuart Charters. Guidelines for Performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.
- [26] Edmund Clarke, Daniel Kroening, and Flavio Lerda. *A Tool for Checking ANSI-C Programs*, page 168–176. Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 2004. ISBN 9783540247302. doi: 10.1007/978-3-540-24730-2_15.
- [27] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011)*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190, Snowbird, UT, USA, 2011. Springer. doi: 10.1007/978-3-642-22110-1_16.
- [28] Matthias Heizmann, Daniel Dietsch, Vincent Langenfeld, and Andreas Podelski. Ultimate Automizer with array interpolation. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013)*, volume 7795 of *Lecture Notes in Computer Science*, pages 455–457, Rome, Italy, 2013. Springer.
- [29] Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. 2LS for program analysis. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016)*, volume 9636 of *Lecture Notes in Computer Science*, pages 98–104, Eindhoven, The Netherlands, 2016. Springer. doi: 10.1007/978-3-662-49674-9_6.
- [30] Marek Chalupa, Jan Strejček, and Martin Višňovský. Symbiotic 7: Integration of a new slicer. In *Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2020)*, volume 12079 of *Lecture Notes in Computer Science*, pages 413–417, Dublin, Ireland, 2020. Springer. doi: 10.1007/978-3-030-45237-7_27.
- [31] Jiří Barnat et al. DiVinE 3.0 – an explicit-state LTL model checker. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013)*, volume 8044 of *Lecture Notes in Computer Science*, pages 863–868, Saint Petersburg, Russia, 2013. Springer. doi: 10.1007/978-3-642-39799-8_60.
- [32] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: A framework for abstraction refinement-based model checking. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design (FMCAD 2017)*, pages 176–179, Vienna, Austria, 2017. IEEE. doi: 10.23919/fmcad.2017.8102257.
- [33] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Alberto Griggio. Verifying dynamic trait objects in Rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP 2022)*, pages 321–330, Pittsburgh, PA, USA, 2022. ACM. doi: 10.1145/35110457.3513031.
- [34] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV 2015)*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361, San Francisco, CA, USA, 2015. Springer. doi: 10.1007/978-3-319-21690-4_20.
- [35] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 200)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Budapest, Hungary, 2008. Springer. doi: 10.1007/978-3-540-78800-3_24.
- [36] Aina Niemetz and Mathias Preiner. Bitwuzla. In *Computer Aided Verification*, page 3–17, Cham, Switzerland, 2023. Springer Nature Switzerland. ISBN 9783031377037. doi: 10.1007/978-3-031-37703-7_1.

- [37] Alessandro Cimatti, Alberto Griggio, Bastiaan J. Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013)*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107, Rome, Italy, 2013. Springer. doi: 10.1007/978-3-642-36742-7_7.
- [38] Haniel Barbosa et al. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2022)*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442, Munich, Germany, 2022. Springer. doi: 10.1007/978-3-030-99524-9_24.
- [39] Bruno Dutertre. Yices 2.2. In *Proceedings of the 26th International Conference on Computer-Aided Verification (CAV 2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744, Vienna, Austria, 2014. Springer. doi: 10.1007/978-3-319-08867-9_49.
- [40] Edmund M. Clarke and E. Allen Emerson. *Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic*, page 52–71. Springer-Verlag, Berlin, Heidelberg, Germany, 1982. ISBN 354011212X. doi: 10.1007/bfb0025774.
- [41] Edmund M. Clarke. *Model checking*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. ISBN 9783540696599. doi: 10.1007/bfb0058022.
- [42] Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C–35(8):677–691, August 1986. ISSN 2326-3814. doi: 10.1109/tc.1986.1676819.
- [43] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. *Model Checking and the State Explosion Problem*, page 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 2012. ISBN 9783642357466. doi: 10.1007/978-3-642-35746-6_1.
- [44] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000)*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125, Austin, TX, USA, 2000. Springer. doi: 10.1007/3-540-40922-X_8.
- [45] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. *Software Verification Using k-Induction*, page 351–368. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 9783642237027. doi: 10.1007/978-3-642-23702-7_26.
- [46] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006. doi: 10.1145/1217856.1217859.
- [47] Daniel Kroening and Ofer Strichman. *Decision Procedures*. Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 2016. ISBN 9783662504970. doi: 10.1007/978-3-662-50497-0.
- [48] Robert Brummayer and Armin Biere. *Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays*, page 174–177. Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 2009. ISBN 9783642007682. doi: 10.1007/978-3-642-00768-2_16.
- [49] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. Boolean Satisfiability Solvers and Their Applications in Model Checking. *Proceedings of the IEEE*, 103(11):2021–2035, November 2015. ISSN 1558-2256. doi: 10.1109/jproc.2015.2455034.
- [50] Rajdeep Mukherjee, Michael Tautschnig, and Daniel Kroening. *v2c – A Verilog to C Translator*, page 580–586. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. ISBN 9783662496749. doi: 10.1007/978-3-662-49674-9_38.
- [51] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, pages 137–148, Auckland, New Zealand, 2009. IEEE. doi: 10.1109/ASE.2009.17.
- [52] Erika Ábrahám and Klaus Havelund, editors. *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5–13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, 2014. Springer Berlin Heidelberg. ISBN 9783642548628. doi: 10.1007/978-3-642-54862-8.
- [53] Mikhail R. Gadelha, Rafael S. Menezes, and Lucas C. Cordeiro. ESBMC 6.1: Automated Test Case Generation Using Bounded Model Checking. *International Journal on Software Tools for Technology Transfer*, 23(6): 857–861, May 2020. ISSN 1433-2787. doi: 10.1007/s10009-020-00571-2.

- [54] Rafael Menezes et al. ESBMC v7.7: Automating Branch Coverage Analysis Using CFG-Based Instrumentation and SMT Solving. In *Proceedings of the 28th International Conference on Fundamental Approaches to Software Engineering (FASE 2025)*, Lecture Notes in Computer Science, Hamilton, New Zealand, 2025. Springer. doi: 10.1007/978-3-031-90900-9_15.
- [55] Lucas C. Cordeiro, Bernd Fischer, and João Marques-Silva. Verifying Multi-Threaded Software Using SMT-Based Context-Bounded Model Checking. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 331–340, Honolulu, HI, USA, 2011. ACM. doi: 10.1145/1985793.1985839.
- [56] Raimundo Barreto, Lucas Cordeiro, and Bernd Fischer. Verifying Embedded C Software with Timing Constraints Using an Untimed Bounded Model Checker. In *2011 Brazilian Symposium on Computing System Engineering*, page 46–52, Los Alamitos, CA, USA, November 2011. IEEE. doi: 10.1109/sbesc.2011.19.
- [57] Mikhail Y. R. Gadelha, Lucas C. Cordeiro, and Denis A. Nicole. Encoding Floating-Point Numbers Using the SMT Theory in ESBMC: An Empirical Evaluation over the SV-COMP Benchmarks. In *Proceedings of the 22nd International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2017)*, volume 10471 of *Lecture Notes in Computer Science*, pages 91–106, Turin, Italy, 2017. Springer. doi: 10.1007/978-3-319-70848-5_7.
- [58] Franz Brauße, Fedor Shmarov, Rafael Menezes, Mikhail R. Gadelha, Konstantin Korovin, Giles Reger, and Lucas C. Cordeiro. ESBMC-CHERI: Towards Verification of C Programs for CHERI Platforms with ESBMC. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*, pages 773–776, Seoul, South Korea, July 2022. ACM. doi: 10.1145/3533767.3543289.
- [59] Rafael Sá Menezes. *GOTO: a verification framework for CProver tools*. Phd thesis, The University of Manchester, June 2025. URL https://ssvlab.github.io/lucasccordeiro/supervisions/phd_thesis_rafael.pdf.
- [60] Mostafijur Rahman Akhond, Saikat Chakraborty, and Gias Uddin. LLM For Loop Invariant Generation and Fixing: How Far Are We?, 2025.
- [61] João F. Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. Smartbugs: A framework to analyze solidity smart contracts, 2020. URL <https://arxiv.org/abs/2007.04771>.
- [62] Darren Cofer, Andrew Gacek, John Backes, Michael W. Whalen, Lee Pike, Adam Foltzer, Michal Podhradsky, Gerwin Klein, Ihor Kuz, June Andronick, Gernot Heiser, and Douglas Stuart. A Formal Approach to Constructing Secure Air Vehicle Software. *Computer*, 51(11):14–23, November 2018. ISSN 1558-0814. doi: 10.1109/mc.2018.2876051.
- [63] Norbert Tihanyi, Tamas Bisztray, Ridhi Jain, Mohamed Amine Ferrag, Lucas C. Cordeiro, and Vasileios Mavroeidis. The FormAI Dataset: Generative AI in Software Security through the Lens of Formal Verification. In *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2023)*, pages 33–40, San Francisco, CA, USA, 2023. ACM. doi: 10.1145/3617555.3617874.
- [64] Norbert Tihanyi, Tamas Bisztray, Mohamed Amine Ferrag, Bilel Cherif, Richard A. Dubniczky, Ridhi Jain, and Lucas C. Cordeiro. Vulnerability Detection: From Formal Verification to Large Language Models and Hybrid Approaches: A Comprehensive Overview, 2025.
- [65] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, Long Beach, CA, USA, 2017. Curran Associates, Inc. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.
- [66] Tom Brown et al. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901, Vancouver, Canada, 2020. Curran Associates, Inc. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf.
- [67] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the Keyboard? Assessing the Security of GitHub Copilot’s Code Contributions. *Communications of the ACM*, 68(2): 96–105, January 2025. ISSN 1557-7317. doi: 10.1145/3610721.
- [68] Xiaowei Huang, Wenjie Ruan, Wei Huang, Gaojie Jin, Dong, et al. A Survey of Safety and Trustworthiness of Large Language Models through the Lens of Verification and Validation, 2023.
- [69] Bernhard Beckert, Jonas Klamroth, Wolfram Pfeifer, Patrick Röper, and Samuel Teuber. *Towards Combining the Cognitive Abilities of Large Language Models with the Rigor of Deductive Program Verification*, page

- 242–257. Springer Nature Switzerland, Cham, Switzerland, October 2024. ISBN 9783031753879. doi: 10.1007/978-3-031-75387-9_15.
- [70] Haoze Wu, Clark W. Barrett, and Nina Narodytska. Lemur: Integrating Large Language Models in Automated Program Verification. In *The Twelfth International Conference on Learning Representations (ICLR 2024)*, page 7652, Vienna, Austria, 2024. OpenReview.net. URL <https://openreview.net/forum?id=Q3YaCghZNt>.
 - [71] Lucas C. Cordeiro. Lucas Cordeiro — Systems and Software Verification Laboratory (SSVLab) Personal Page. <https://ssvlab.github.io/lucasccordeiro/>, 2025. Accessed: May 2026.
 - [72] UK Research and Innovation (UKRI). R&D Investments Spearhead Push to Block Cyber Security Attacks (Soteria consortium, £5.8M). <https://www.openaccessgovernment.org/rd-investments-spearhead-cyber-security-defence/96958/>, 2021.
 - [73] European Commission. ELEGANT — Towards Attestable and Trustworthy Internet-of-Things Infrastructures (H2020 grant agreement No. 957286). CORDIS — EU Research Results. <https://cordis.europa.eu/project/id/957286>, 2021. Accessed: May 2026.
 - [74] UK Research and Innovation (UKRI). Digital Security by Design (DSbD) Programme. <https://www.ukri.org/what-we-do/browse-our-areas-of-investment-and-support/digital-security-by-design/>, 2019. Accessed: May 2026.
 - [75] University of Manchester Research Explorer. VeriBee: Source Code Security (Research Impact Record). <https://research.manchester.ac.uk/en/impacts/veribee-source-code-security/>, 2025. Accessed: May 2026.
 - [76] SecurityBrief UK. UK SMEs Face Rise in Cyber Attacks with Average Cost £7,960. <https://securitybrief.co.uk/story/uk-smes-face-rise-in-cyber-attacks-with-average-cost-gbp-7-960>, 2024. Accessed: May 2026.
 - [77] Gregory Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical Report Planning Report 02-3, National Institute of Standards and Technology (NIST), Gaithersburg, MD, 2002. URL <https://www.nist.gov/document/report02-3pdf>. Accessed: May 2026.
 - [78] Consortium for Information and Software Quality. The cost of poor software quality in the US: A 2020 report. Technical report, CISQ, 2020. URL <https://www.it-cisq.org/the-cost-of-poor-software-quality-in-the-us-a-2020-report/>.
 - [79] Department for Science, Innovation and Technology. Cyber security breaches survey 2024. Technical report, UK Government, 2024. URL <https://www.gov.uk/government/statistics/cyber-security-breaches-survey-2024>.
 - [80] IBM Systems Sciences Institute. Relative Cost of Fixing Defects. Black Duck <https://www.blackduck.com/blog/cost-to-fix-bugs-during-each-sdlc-phase.html>, 1981.
 - [81] CNN Business / Parametrix. CrowdStrike Outage: Cost and Cause (July 2024). CNN Business: <https://www.cnn.com/2024/07/24/tech/crowdstrike-outage-cost-cause>; Altitudes Magazine: <https://www.theguardian.com/technology/article/2024/jul/24/crowdstrike-outage-companies-cost>, 2024.
 - [82] DefiLlama. DeFi Total Value Locked (TVL) Dashboard. <https://defillama.com/>, 2025. Accessed: May 2026.
 - [83] The Block. Over \$115 Billion of Ether Is Now Staked on the Beacon Chain. <https://www.theblock.co/post/280535/over-115-billion-of-ether-is-now-staked-on-the-beacon-chain>, 2024. Accessed: May 2026.
 - [84] Halborn Security. The Top 100 DeFi Hacks Report 2025. <https://www.halborn.com/reports/top-100-defi-hacks-2025>; summary at <https://www.halborn.com/blog/post/halborn-all-time-top-100-defi-hacks-report-summary>, 2025. Accessed: May 2026.
 - [85] Chainalysis. 2025 Crypto Crime Trends. <https://www.chainalysis.com/blog/2025-crypto-crime-report-introduction/>, 2025. Accessed: May 2026.
 - [86] CoinDesk. Axie Infinity’s Ronin Network Suffers \$625M Exploit. <https://www.coindesk.com/tech/2022/03/29/axie-infinitys-ronin-network-suffers-625m-exploit>, 2022. Accessed: May 2026.
 - [87] Ethereum Foundation. Ethereum Bug Bounty Program. <https://ethereum.org/bug-bounty/>; CryptoRank: <https://cryptorank.io/news/feed/f4381-ethereum-foundation-bug-bounty-million>, 2025. Accessed: May 2026.

- [88] Avionics Today / Vita Technologies. DO-178C: Certification Cost-Effective Avionics Systems. <https://vita.militaryembedded.com/2325-do-178c-certification-cost-effective-avionics-systems/>, 2017. Accessed: May 2026.
- [89] Better Embedded Systems Blog / Eurocontrol. Cost of Highly Safety-Critical Software. <https://betterembsw.blogspot.com/2018/10/cost-of-highly-safety-critical-software.html>, 2018. Accessed: May 2026.
- [90] Statista. Lockheed Martin Annual R&D Expenditure (2023). <https://www.statista.com/statistics/268928/expenditure-on-research-and-development-of-defense-supplier-lockheed-martin/>, 2024. Accessed: May 2026.
- [91] Tong Wu, Shale Xiong, Edoardo Manino, Gareth Stockwell, and Lucas C. Cordeiro. Verifying components of arm(r) confidential computing architecture with esbmc, 2025. arXiv preprint first posted June 2024 (arXiv:2406.04375).
- [92] Arm Holdings plc. Arm Holdings Annual Revenue (FY2025). MacroTrends / StockAnalysis: <https://stockanalysis.com/stocks/arm/market-cap/>, 2025. Accessed: May 2026.
- [93] IBM and Ponemon Institute. Cost of a Data Breach Report 2024. <https://newsroom.ibm.com/2024-07-30-ibm-report-escalating-data-breach-disruption-pushes-costs-to-new-highs>, 2024. Accessed: May 2026.
- [94] Embedded Computing Design / Embedded.com. A Sensible Solution for Addressing the CVE Explosion in IoT Devices. <https://www.embedded.com/a-sensible-solution-for-addressing-the-cve-explosion-in-iot-devices/>, 2024. Accessed: May 2026.
- [95] TuxCare. The Real-World Cost of Not Patching a Critical CVE: A Security ROI Perspective. <https://tuxcare.com/blog/the-real-world-cost-of-not-patching-a-critical-cve-a-security-roi-perspective/>, 2024. Accessed: May 2026.
- [96] Sibros. The Current State of Automotive Software-Related Recalls. <https://www.sibros.tech/post/the-current-state-of-automotive-software-related-recalls>, 2024. Accessed: May 2026.
- [97] Zealynx Security. Smart Contract Audit Cost in 2025: What You Need to Know. <https://www.zealynx.io/research/audit-ops/audit-pricing-2026>, 2025. Accessed: May 2026.
- [98] Software Secured. Is the Price Always Right? A Comprehensive Guide to Penetration Testing Costs. <https://www.softwaresecured.com/post/is-the-price-always-right-a-comprehensive-guide-to-penetration-testing-costs>, 2024. Accessed: May 2026.
- [99] IEEE Spectrum / Wikipedia. Boeing 737 MAX Groundings (March 2019 – December 2020). IEEE Spectrum: <https://spectrum.ieee.org/how-the-boeing-737-max-disaster-looks-to-a-software-developer>; note = Accessed: May 2026, 2021.
- [100] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. *Enchanting Program Specification Synthesis by Large Language Models Using Static Analysis and Program Verification*, page 302–328. Springer Nature Switzerland, Cham, Switzerland, 2024. ISBN 9783031656309. doi: 10.1007/978-3-031-65630-9_16.
- [101] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models, 2022.
- [102] Marco Lewis, Sadegh Soudjani, and Paolo Zuliani. Formal verification of quantum programs: Theory, tools, and challenges. *ACM Transactions on Quantum Computing*, 5(1):1–35, December 2023. ISSN 2643-6817. doi: 10.1145/3624483.
- [103] Simon J. Gay, Rajagopal Nagarajan, and Nikolaos Papanikolaou. *QMC: A Model Checker for Quantum Systems*, page 543–547. Springer Berlin Heidelberg, Princeton, NJ, USA, 2008. ISBN 9783540705451. doi: 10.1007/978-3-540-70545-1_51.
- [104] Mingkuan Xu et al. Quartz: superoptimization of quantum circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI ’22*, page 625–640, San Diego, CA, USA, June 2022. ACM. doi: 10.1145/3519939.3523433.
- [105] Felipe R. Monteiro, Erickson H. da S. Alves, Isabela S. Silva, Hussama I. Ismail, Lucas C. Cordeiro, and Eddie B. de Lima Filho. Esbmc-gpu a context-bounded model checking tool to verify cuda programs. *Science of Computer Programming*, 152:63–69, January 2018. ISSN 0167-6423. doi: 10.1016/j.scico.2017.09.005.

- [106] Arut Prakash Kaleeswaran, Arne Nordmann, Thomas Vogel, and Lars Grunske. A user study for evaluation of formal verification results and their explanation at bosch. *Empirical Software Engineering*, 28(5), September 2023. ISSN 1573-7616. doi: 10.1007/s10664-023-10353-4.
- [107] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE'16, page 332–343, Singapore, August 2016. ACM. doi: 10.1145/2970276.2970347.