VO-GCSE: Verification Optimization through Global Common Subexpression Elimination

Rafael Sá Menezes[*,§], Norbert Tihanyi[†], Ridhi Jain[†], Alexander Levin[‡], Rosiane de Freitas[§], and Lucas C. Cordeiro[*,§]
[*]The University of Manchester, UK, [†]Technology Innovation Institute, UAE, [‡]Nvidia, USA, [§]UFAM, Brazil

ABSTRACT

Compiler optimizations enhance machine code efficiency while maintaining functionality. Global Common Subexpression Elimination (GCSE) removes redundant expressions using Available Expression (AE) data flow based on points-to-analysis. Despite its success in compilers such as GCC and LLVM, GCSE remains unexplored in Formal Verification (FV) due to domain-specific challenges. This paper is the first to introduce *"Verification Optimization through GCSE"* (VO-GCSE), a GCSE algorithm designed for FV, considering symbolic memory models and the unique attributes of FV tools. We integrated VO-GCSE into the Efficient SMT-based Context-Bounded Model Checker (ESBMC), leveraging its existing logic-based infrastructure, including Value-Set Analysis (VSA) and Abstract Interpretation (AI). We transformed AE data flow into an AI challenge and simplified VSA into points-to-analysis. VO-GCSE was evaluated using three diverse sources: the SV-COMP benchmarks, the FormAI dataset comprising 330, 000 C samples, and real-world verification cases from Intel®. The experimental results show that (a) VO-GCSE reduces verification time for programs with similar dereferences, and (b) AE computation through AI does not complicate the analysis. Video: https://www.youtube.com/watch?v=6QKGcDfp5is

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

Formal Verification, Bounded Model Checking, Compiler Optimization, GCSE

ACM Reference Format:
Rafael Sá Menezes[*,§], Norbert Tihanyi[†], Ridhi Jain[†], Alexander Levin[‡], Rosiane de Freitas[§], and Lucas C. Cordeiro[*,§]. 2025. VO-GCSE: Verification Optimization through Global Common Subexpression Elimination. In . ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Compiler optimization techniques improve the efficiency and performance of generated machine code without altering its functionality [25]. These optimization techniques focus on reducing code size, boosting execution speed, or enhancing memory efficiency [21]. Common compiler optimization techniques include *Inlining* [27], which replaces function calls with the actual function body to reduce call overhead, *Dead Code Elimination* (DCE) [3], focused on removing code segments that do not impact the program's outcome, or *Constant Folding* [10], which entails evaluating constant expressions at compile time. Furthermore, the technique of *Operator Strength Reduction*[11] involves substituting resource-intensive operations with more cost-effective but functionally equivalent ones or *Global Value Numbering* (GVN) [19], optimizing redundant calculations throughout the program. *Loop invariants* [16, 26] facilitate inductive verification; however, they have limited applicability and high overhead. Loop unwinding and Global Common Subexpression Elimination (GCSE) are compiler optimization techniques designed to enhance program performance [4]. GCSE, introduced by Cocke in 1970 [9], reduces redundant calculations by identifying and replacing repeated expressions across basic blocks. Although loop unwinding is integral to Formal Verification (FV), particularly in Bounded Model Checking (BMC) [8], GCSE has not been adopted in this specific domain before. Methods such as BMC and Abstract Interpretation (AI) [15] are commonly used to verify critical software components [30], with BMC reducing false positives by generating concrete counterexamples. However, it can increase the verification time due to the extra effort required to reach a fixed-point [14]. In such cases, integrating GCSE could accelerate the verification process, particularly in memory-safety verifications.

This paper is the first to introduce *Verification Optimization through Global Common Subexpression Elimination (VO-GCSE)*, a novel framework integrating GCSE into FV, specifically within the *Efficient SMT-based Context-Bounded Model Checker* (ESBMC) [17]. To the best of our knowledge, this marks the first application of GCSE in FV, significantly improving verification time in critical domains such as firmware, Linux kernels, and embedded systems. Figure 1 illustrates the *VO-GCSE* framework's architecture within ESBMC. The motivation for incorporating GCSE into the verification arose from practical challenges in an industrial project. In contrast to traditional compiler optimizations, FV demands additional constraints to ensure semantic preservation while maintaining a unique exploration tree, complicating GCSE adaptation. The primary scientific contributions of this paper are as follows.

(1) Adapting the GCSE algorithm for formal software verification, considering factors such as symbolic models and the unique capabilities of various verification tools;
(2) Integration of *VO-GCSE* into the SMT-based model checker, ESBMC (publicly available since version 7.5.0);
(3) An AI domain to compute AE over programs. This algorithm considers symbolic memory models for its analysis, resulting in more suitable results for FV tools;
(4) Experimental evaluation of the *VO-GCSE* algorithm on the Software Verification Competition (SV-COMP) [7], demonstrating performance gains in memory verification;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FSE'25, June 2025, Trondheim, Norway
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(5) Comparative testing of ESBMC with and without *VO-GCSE*, ensuring consistency and robustness in results.

2 MOTIVATION

Software verification guarantees that a program satisfies a given safety property across all execution paths [5], in contrast to software testing, which only provides correctness based on sampled cases. In industry, FV methods such as BMC are widely used, with ESBMC being a prominent tool to verify C and C++-based software components, including firmware and embedded systems [17, 23, 24, 30].

During a real system analysis, Intel® developers observed that modifying a specific type of C code significantly improved software verification time and memory usage, enabling the verification of embedded components that were previously unattainable. Initially, all modifications were applied manually, but this eventually led to a collaboration between ESBMC developers and Intel domain experts, which resulted in the identification of further optimizations and automation of the process to improve verification efficiency. Although some of these optimizations are domain-specific, some (e.g., loop unfolding and instruction reordering [3]) could be applied in general. A notable example of efficient optimization was using an intermediate variable to cache dereferences. For a practical illustration, refer to the real-life application code fragment in Listing 1 as a motivating example.

```
typedef struct { unsigned Flags;} Aux;
typedef struct { Aux Aux; unsigned Wc; unsigned V;} RegEntry;
typedef struct {RegEntry *Map;} table;
void write(table *tbl, unsigned EntryIndex);
int main() {
    RegEntry e[10000]; table M; M.Map = e;
    for (int i = 0; i < 10000; i++) write(&M, i);
    return 0;   }
void write(table *tbl, unsigned EntryIndex) {
    unsigned Data64 = 42;
    tbl->Map[EntryIndex].Aux.Flags &= 1;
    tbl->Map[EntryIndex].Wc++;
    tbl->Map[EntryIndex].V = Data64;}
```

Listing 1: Motivating Example (Original)

ESBMC's symbolic memory model [12, 13] complicates pointer tracking, as each dereference requires validating pointer safety. The complexity surges with consecutive dereferences. Introducing intermediate variables to store dereference results simplifies this process, dramatically accelerating verification. Common in hardware-specific C programs, this approach, combined with GCSE, could deliver substantial benefits in various scenarios. For instance, our motivating example contains multiple dereferences of `tbl`. For each dereference, ESBMC will do the full target computation from scratch (which also adds the same assertions). The unoptimized `write` function repeatedly dereferences `tbl`, leading to redundant target computations.

```
void write_optimized(table *tbl, unsigned EntryIndex) {
    unsigned Data64 = 42;
    RegEntry *pMap = &(tbl->Map[EntryIndex]);
    pMap->Auxiliary.Flags &= 1; pMap->Wc++; pMap->V = Data64;}
```

Listing 2: Motivating Example (Optimized write function)

Optimizing it with an intermediate variable improves efficiency, as the unoptimized version took 105 seconds for symbolic execution with 270015 assignments using ESBMC 7.8.1. Replacement of the `write` function with the `write_optimized` function (see Lisiting 2) allows the verification to be completed in less than 7 seconds —a 1500% speedup. This small example highlights the effectiveness of GCSE optimization in eliminating redundant calculations.

The optimization was initially applied manually, but this approach can be automated by calculating AE [18] through data-flow analysis and modifying the Control-flow Graph (CFG) [3]. However, not all verifiers, including ESBMC, have a data-flow solver. ESBMC uses an Abstract Interpreter[15] for computing program properties. Implementing *VO-GCSE* with the Abstract Interpreter, without a data-flow solver, presents challenges, which we explore in this research and the associated experimental results.

3 RESEARCH DESIGN AND METHODOLOGY

The optimization requires two key tasks: (1) computing AE in GOTO language [12, 22] (ESBMC's intermediate representation), and (2) applying GCSE to the GOTO code while preserving its original semantics. A prototype of the *VO-GCSE* algorithm has been developed and successfully integrated into ESBMC since version 7.5.

3.1 Computing AE

We use value-set analysis (VSA) as a static pointer analysis to compute AE, identifying which expressions become unavailable due to dereferencing. In the following code snippet: `int sum = x + 42; int *ptr = &x; *ptr = 0; sum = x + 42;`, a cached subexpression $x + 42$ is invalidated by the dereference operation $*ptr = 0;$. The analysis considers the set of all possible expression combinations in the program, i.e., $(\mathcal{P}(\mathbb{E}))$. Although this domain can reach exponential sizes, we only store hashes of the expression (ESBMC contains an internal SHA256 to hash its expressions). In this domain, the *infimum* is the empty set, while the *supremum* is all the expressions in the program. However, computing the *supremum* is unnecessary, as we can always check whether the set has grown with a new expression. Nontrivial expressions (e.g., $x + 42$) are recursively added to the domain, while trivial expressions (e.g., x) are excluded. Expressions dependent on invalidated pointers are removed due to the values being indirectly updated.

3.2 Applying GCSE

Optimization occurs in two key steps: (1) identifying the maximum available sub-expression for each program statement and (2) initializing expressions. This process is applied to each GOTO function [22], as illustrated in Figure 3.

3.2.1 Obtaining Max sub-expressions. Expressions are generated by iterating through function statements, recursively evaluating operands against the AE set until an expression is found or the empty expression is reached. Sub-expressions are then instrumented into new temporary variables, declared but not initialized. If the LHS is a dereference, the intermediate variable is initialized for assignment. The decision on when to initialize relies on program-specific details, such as recomputed expressions (e.g., in loops) or those initialized late due to dereferences. Initialization occurs if the

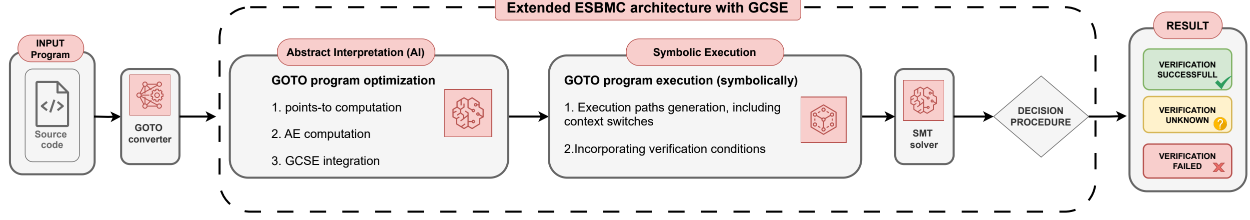

Figure 1: VO-GCSE: An Enhanced ESBMC Architecture with GCSE for Safety Evaluation Using a *Decision Procedure* [20].

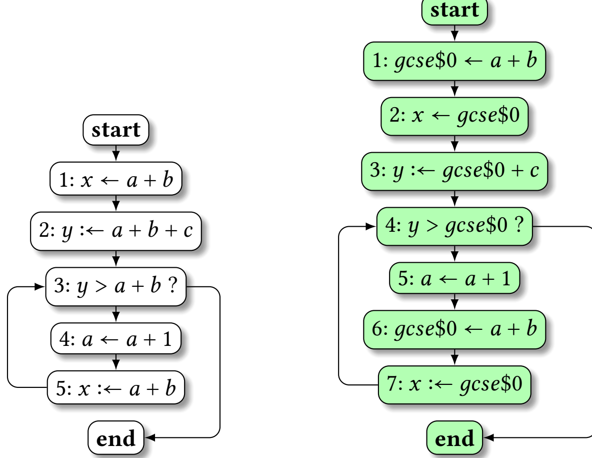

Figure 2: Original **Figure 3: GCSE optimized**

sub-expression was unavailable in the previous statement, triggering an assignment.

3.2.2 Short-circuit expressions. In C, boolean expressions can short-circuit from the left. This may be exploited by creating expressions that can only be computed if the previous condition was met, e.g., `ptr != NULL && *ptr == 42`. This behavior affects the GCSE algorithm, as precomputing expressions can cause errors. Thus, we only make the first operand (i.e., the leftmost one) available.

4 EXPERIMENTAL EVALUATION

We conducted a series of experiments to measure the effectiveness of the *VO-GCSE* algorithm for software analysis and answer the following experimental questions:

EQ1: Does *VO-GCSE* algorithm allows ESBMC to verify unique benchmarks?

EQ2: What types of benchmarks does the *VO-GCSE* causes improvement or degradation?

Our analysis includes two distinct evaluations: a performance evaluation using the Software Verification Competition (SV-COMP) [7] benchmark, consisting of 9,340 test-cases across 19 sub-categories, and an implementation evaluation on the FormAI-v2 dataset [28], which contains 330, 000 C samples.

4.1 Implementation Verification using FormAI

The FormAI dataset [28, 29] does not contain many cases where the GCSE algorithm is particularly advantageous. However, its 330, 000 C samples are highly effective for identifying implementation bugs when discrepancies arise between verification results with and without GCSE. We uncovered three bugs in ESBMC 7.5.0 VO-GCSE that led to segmentation faults stemming from missing exception handling for failed VSA computations. Across the 330, 000 samples, 195 cases (0.059%) showed discrepancies, leading to a manual review. In 103 cases, verification without GCSE timed out (120 s on an Intel® Xeon® Platinum8375C 32-core CPU with 128, GBRAM), whereas GCSE finished on time. In seven cases, Boolector crashed, but switching to Z3 yielded consistent results; in 75, GCSE slightly increased verification time. The remaining 10 cases were the most interesting: verification passed without GCSE yet failed with GCSE, demanding a deeper investigation. We used bounded model checking with loop unwinding set to 1; without GCSE, counterexamples beyond this bound are labeled "SUCCESSFUL," but GCSE introduces intermediate variables that lower the bound, exposing hidden bugs. After we fixed the VSA computation issues (since ESBMC7.8.1), no further discrepancies emerged in verification outcomes.

4.2 Performance Analysis on SV-COMP

Tests on the FormAI dataset focused on identifying discrepancies in verification results or implementation errors. However, a robust performance analysis was required to address our research questions and assess scenarios where GCSE offers advantages. Therefore, we used the SV-COMP benchmark [7]. The benchmarks were executed using benchexec [6] with the below configuration:

- 32-core vCPU with the single-threaded operation. Both ESBMC and the SMT solver do not utilize multiple cores, therefore, each thread handles a separate verification task;
- 120 seconds timeout, based on SVCOMP-23 results [7], where 95% of benchmarks were solved within this time;

The SV-COMP benchmarks assess verifiers against common vulnerabilities, such as memory corruption, arithmetic overflows, and assertion failures. Here, we selected all Memory and Reachability categories, excluding concurrency-related tasks because the current implementation is not thread-aware. Our results, summarized in Table 1, are based on 9340 benchmarks across 19 subcategories. C_T and C_T^{GCSE} denote correct true results (no violation found), C_F and C_F^{GCSE} denote correct false results (a violation is found), I_T and I_T^{GCSE} denote incorrect true results, I_F and I_F^{GCSE} denote incorrect false results. The table categorizes the benchmarks by Category (type of property being verified) and Sub-Category (type of benchmarks). The dataset is imbalanced, with categories such as *Memsafety-Other*, containing 38 benchmarks, compared to *Memsafety-Juliet* with 2828. To account for this, SV-COMP normalizes scores across all sub-categories. Enabling VO-GCSE resulted in an overall time increase of about 3%, while categories with performance gains showed a 3% reduction. Notably, *Reachability-Arrays* experienced a **52% improvement** in average for verification time (from start to verdict). Some gains were also observed in categories such as *Memory-Heap*, *Arrays*, *BitVectors*, and *Recursive*.

Table 1: Aggregated Results for the SV-COMP benchmark verification.

Category	**Sub-Category**	**Quantity**	C_T	C_F	I_T	I_F	C_T^{GCSE}	C_F^{GCSE}	I_T^{GCSE}	I_F^{GCSE}	CPU(s)	CPU(s)-GCSE
Memory	Arrays	43	0	18	0	1	0	18	0	1	5.81	5.83
Memory	Heap	153	38	76	0	1	38	76	0	1	316	314
Memory	LinkedLists	79	25	27	0	0	25	27	0	0	242	249
Memory	Other	38	2	19	0	0	2	19	0	0	130	143
Memory	Juliet	2828	1438	1390	0	0	1438	1390	0	0	9770	9880
Memory	MemCleanup	61	1	59	0	0	1	59	0	0	68.4	69.4
Reachability	Arrays	300	11	74	0	0	10	74	0	0	257	169
Reachability	BitVectors	49	19	13	0	0	19	13	0	0	165	163
Reachability	ControlFlow	22	12	6	0	0	12	6	0	0	216	236
Reachability	ECA	1263	305	108	0	0	311	102	0	0	8640	8540
Reachability	Floats	434	356	32	0	0	356	33	0	0	1490	1460
Reachability	Heap	159	45	59	0	0	45	59	0	0	252	222
Reachability	Loops	685	218	142	0	0	218	143	0	1	4060	4080
Reachability	ProductLines	597	170	263	0	0	170	263	0	0	685	704
Reachability	Recursive	59	8	22	0	0	8	22	0	0	196	186
Reachability	Sequentialized	563	32	133	0	0	30	129	4	0	1110	1250
Reachability	XCSP	114	50	44	0	0	50	44	0	0	369	358
Reachability	Combinations	671	19	249	0	0	18	245	0	0	5240	5640
Reachability	Hardware	1222	83	169	0	0	83	163	0	0	7030	7710

4.3 Discussion

The efficiency of GCSE is assessed by comparing verification results with and without it, considering its semantic equivalence, impact on the solver, and optimization cost. To address the experimental questions, we analyze benchmarks affected positively or negatively and compare preprocessing times for the optimized GOTO program.

Additionally, some benchmarks in the Hardware and ECA categories incurred a 5× time penalty due to unbounded loops with thousands of lines. These loops caused multiple interactions during AI, increasing time and memory. This limitation arises from ESBMC's AI handling on a per-statement basis, rather than at the CFG level. Generally, AI remains effective in computing AE. Significant performance gains were observed in the Arrays sub-category of reachability (c.f. Table 1), where the optimization reduced total time, especially during safety proofs. Similarly, the category with multiple dereferences to the same array position had reduced total time. This is likely due to the combination of ESBMC's Invariant Generation and k-induction. While ECA and Hardware had more timeouts, the optimization allowed the verification of new benchmarks that ESBMC previously failed to verify.

- **EQ1**: Yes, it allows ESBMC to verify programs that it previously could not due to computational constraints. When looking at the verdicts themselves, the method just accelerated ESBMC in finding them. The main limitation and difficulty of the method regarding soundness is the previous pointer analysis of ESBMC, which can cause crashes and limit the effectiveness.
- **EQ2**: The optimization works the best when the benchmarks manipulate pointers predictably (e.g., struct initialization), specially if the pointer operand is constant. The worst case occurs in functions on unbounded loops with thousands of statements due to how long the pointer analysis takes. For example, the hardware category emulates bitwise operations that change the state of a system. These hardware operations contain thousands of variables changed in an unbounded loop with thousands of statements.

Our analysis shows that *VO-GCSE* introduces a 3% increase in overall verification time, primarily due to delays in the Hardware category. After an initial human-led verification analysis, this overhead can be mitigated by selectively disabling *VO-GCSE*. As a side note, even in compilers like GCC or Clang, determining whether to enable optimization levels from $O0$ to $O3$ and identifying which level performs better remains an open problem and is not a trivial task to resolve. Despite the time increase, *VO-GCSE* delivers significant performance improvements— **up to** 52%**—in memory-related verification tasks (i.e., makes heavy use of pointer dereference), enabling previously infeasible verifications**, as demonstrated in our motivating example. The AI method effectively computes AE, and benchmark results confirm that *VO-GCSE* preserves the verification status, ensuring soundness.

4.3.1 *Threats to Validity.*

- Although we conducted rigorous testing using the FormAI dataset and SV-COMP to address potential implementation issues, no code can be guaranteed to be entirely bug-free. We encourage the research community to report any issues by opening a new ticket on the ESBMC GitHub page.
- ESBMC offers several optimizations, and combining GCSE with techniques like Interval Analysis can yield stronger invariants through interval propagation. Further behavioral analysis using various methods will be carried out to evaluate the potential of GCSE across different categories.

4.3.2 Availablility. ESBMC with VO-GCSE optimization is available for public access on the ESBMC GitHub webpage [1] along with a real-world-like example [2]. Option flag `--gcse` enables the GCSE optimization within ESBMC (since version 7.5.0). Tool demonstration video: https://www.youtube.com/watch?v=6QKGcDfp5is

5 CONCLUSIONS

We introduced the *VO-GCSE* algorithm to optimize C program verification by capturing AE using an Abstract Domain. It enhances ESBMC's handling of repeated dereferences, yielding significant performance gains, particularly in the SV-COMP Arrays subcategory. The algorithm accelerates reachability verification in programs with frequent dereferences. Our results suggest that *VO-GCSE* could further improve ESBMC's support for Kotlin and C++, which rely on virtual pointers and tables for polymorphism. Future work could explore optimizations such as anticipated expressions, partial redundancy elimination, and constant propagation to assess their impact on verification challenges. Additionally, addressing the challenge of reaching a fixpoint could involve developing heuristics to selectively skip loops based on features like unboundedness, program types, or verification properties. Implementing the Abstract Interpreter at the CFG level could expedite fixpoint computation while balancing precision and performance.

REFERENCES

[1] 2024. https://github.com/esbmc/esbmc/releases/tag/v7.8.1. Accessed: 2025-01-13.

[2] 2024. https://raw.githubusercontent.com/esbmc/esbmc/refs/heads/master/docs/examples/GCSE_real_example.c. Accessed: 2025-01-13.

[3] Aho Alfred V, Lam Monica S, Sethi Ravi, Ullman Jeffrey D, et al. 2007. *Compilers-principles, techniques, and tools.* pearson Education.

[4] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.* 26, 4 (dec 1994), 345–420. https://doi.org/10.1145/197405.197406

[5] David Basin. 2021. *The cyber security body of knowledge.* University of Bristol,, ch. Formal Methods for Security, version..[Online ....

[6] Dirk Beyer. 2016. Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 887–904.

[7] Dirk Beyer. 2023. Competition on software verification and witness validation: SV-COMP 2023. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 495–522.

[8] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded model checking using satisfiability solving. *Formal methods in system design* 19 (2001), 7–34.

[9] John Cocke. 1970. Global Common Subexpression Elimination. *SIGPLAN Not.* 5, 7 (jul 1970), 20–24. https://doi.org/10.1145/390013.808480

[10] Keith D. Cooper, Ken Kennedy, and Linda Torczon. 2003. Compilers. In *Encyclopedia of Physical Science and Technology (Third Edition)* (third edition ed.), Robert A. Meyers (Ed.). Academic Press, New York, 433–442. https://doi.org/10.1016/B0-12-227410-5/00126-5

[11] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. 2001. Operator Strength Reduction. *ACM Trans. Program. Lang. Syst.* 23, 5 (sep 2001), 603–625. https://doi.org/10.1145/504709.504710

[12] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. 2011. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering* 38, 4 (2011), 957–974.

[13] Lucas C. Cordeiro and Bernd Fischer. 2011. Verifying multi-threaded software using smt-based context-bounded model checking. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 331–340.

[14] Patrick Cousot. 2021. *Principles of Abstract Interpretation.* MIT Press.

[15] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* 238–252.

[16] Carlo A Furia, Bertrand Meyer, and Sergey Velder. 2014. Loop invariants: Analysis, classification, and examples. *ACM Computing Surveys (CSUR)* 46, 3 (2014), 1–51.

[17] Mikhail R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. [n. d.]. *ESBMC: 5.0: An Industrial-Strength Model Checker.* https://github.com/esbmc/esbmc

[18] Matthew S Hecht and Jeffrey D Ullman. 1975. A simple algorithm for global data flow analysis problems. *SIAM J. Comput.* 4, 4 (1975), 519–532.

[19] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts) *(POPL '73)*. Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/512927.512945

[20] Daniel Kroening and Ofer Strichman. 2016. *Decision procedures.* Springer.

[21] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[22] Rafael Menezes, Daniel Moura, Helena Cavalcante, Rosiane de Freitas, and Lucas C Cordeiro. 2022. ESBMC-Jimple: verifying Kotlin programs via jimple intermediate representation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis.* 777–780.

[23] Rafael Sá Menezes, Mohannad Aldughaim, Bruno Farias, Xianzhiyu Li, Edoardo Manino, Fedor Shmarov, Kunjian Song, Franz Brauße, Mikhail R. Gadelha, Norbert Tihanyi, Konstantin Korovin, and Lucas C. Cordeiro. 2024. ESBMC v7.4: Harnessing the Power of Intervals. In *Tools and Algorithms for the Construction and Analysis of Systems*, TBA (Ed.). Lecture Notes in Computer Science, Vol. 14572. Springer, 376–380. https://doi.org/10.1007/978-3-031-57256-2_24

[24] Felipe R. Monteiro, Mikhail R. Gadelha, and Lucas C. Cordeiro. 2022. Model checking C++ programs. *Softw. Test. Verification Reliab.* 32, 1 (2022).

[25] Paul B. Schneck. 1973. A Survey of Compiler Optimization Techniques. In *Proceedings of the ACM Annual Conference* (Atlanta, Georgia, USA) *(ACM '73)*. Association for Computing Machinery, New York, NY, USA, 106–113. https://doi.org/10.1145/800192.805690

[26] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Simplifying loop invariant generation using splitter predicates. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23.* Springer, 703–719.

[27] Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. 2022. Understanding and exploiting optimal function inlining. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 977–989. https://doi.org/10.1145/3503222.3507744

[28] Norbert Tihanyi, Tamas Bisztray, Mohamed Amine Ferrag, Ridhi Jain, and Lucas C. Cordeiro. 2025. How secure is AI-generated code: a large-scale comparison of large language models. *Empirical Software Engineering* 30, 47 (2025). https://doi.org/10.1007/s10664-024-10590-1

[29] Norbert Tihanyi, Tamas Bisztray, Ridhi Jain, Mohamed Amine Ferrag, Lucas C. Cordeiro, and Vasileios Mavroeidis. 2023. 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* (San Francisco, CA, USA) *(PROMISE 2023)*. Association for Computing Machinery, New York, NY, USA, 33–43. https://doi.org/10.1145/3617555.3617874

[30] Tong Wu, Shale Xiong, Edoardo Manino, Gareth Stockwell, and Lucas C. Cordeiro. 2024. Verifying components of Arm® Confidential Computing Architecture with ESBMC. *arXiv preprint arXiv:2406.04375* (2024). https://arxiv.org/abs/2406.04375 Accessed: 2024-11-19.

APPENDIX - WALKTHROUGH GUIDELINE

(1) The first step is to download the latest Efficient SMT-based Context-Bounded Model Checker (ESBMC) from the GitHub page. We have released a new version, ESBMC 7.8.1 [1], incorporating all available GCSE fixes and features for this FSE submission. We will use a precompiled binary in this demonstration to avoid installing multiple dependencies. The demonstration will be conducted on Ubuntu 24.04.

```
# Command to update the system and install the build-essential package along with unzip.
sudo apt update && sudo apt upgrade -y && sudo apt install build-essential unzip -y
# Command to download ESBMC
wget https://github.com/esbmc/esbmc/releases/download/v7.8.1/esbmc-linux.zip
#Command to unzip ESBMC
unzip esbmc-linux.zip
#Command to grant execute (+x) permission for ESBMC
chmod +x linux-release/bin/esbmc
```

(2) From the previous STEP, we should have a working ESBMC that we can run

```
# Command to run ESBMC
linux-release/bin/esbmc
```

OUTPUT:

```
ESBMC version 7.8.0 64-bit x86_64 linux
Target: 64-bit little-endian x86_64-unknown-linux with esbmclibc
ERROR: Please provide a program to verify
```

(3) We can compare the performance of running ESBMC with and without GCSE. Our tests evaluate two scenarios: a small motivating example and a real-world one. The real-world example fails verification without GCSE due to an out-of-memory error, specifically on an Intel Xeon Platinum 8375C 32-core CPU with 128 GB of RAM. First, we download the two GCSE examples from the ESBMC GitHub page.

```
# Command to download the two examples: GCSE_real_example.c and GCSE_motivating_example.c

wget https://raw.github.com/esbmc/esbmc/refs/heads/master/docs/examples/GCSE_motivating_example.c
wget https://raw.github.com/esbmc/esbmc/refs/heads/master/docs/examples/GCSE_real_example.c
```

(4) Next, we can run ESBMC on the examples, beginning with the motivating example.

```
# Command to run ESBMC on GCSE_motivating_example.c WITHOUT GCSE
linux-release/bin/esbmc GCSE_motivating_example.c
```

OUTPUT:

```
....
Encoding to solver time: 1.874s
Solving with solver Boolector 3.2.3
Runtime decision procedure: 0.000s
BMC program time: 71.835s

VERIFICATION SUCCESSFUL
```

```
# Command to run ESBMC on GCSE_motivating_example.c WITH GCSE
linux-release/bin/esbmc  GCSE_motivating_example.c --gcse
```

OUTPUT:

```
....
Slicing time: 0.106s (removed 80013 assignments)
Generated 100000 VCC(s), 0 remaining after simplification (1 assignments)
BMC program time: 4.830s

VERIFICATION SUCCESSFUL
```

(5) The difference is clear in the two outputs. Without GCSE, the verification time in our test environment was 71.83 seconds, whereas with GCSE, it was reduced to 4.83 seconds. Similarly, the same performance improvement is observed when testing the `GCSE_real_example.c`. To discover more examples where GCSE can help reduce verification time, one can explore the FormAI dataset and perform the verification both with and without GCSE on more than 330, 000 samples.