Towards a Vertically Integrated Compiler Infrastructure to Improve Solver Time during Symbolic Execution

Sören Tempel
tempel@ibr.cs.tu-bs.de
Technische Universität Braunschweig
Braunschweig, Germany

Christian Dietrich
dietrich@ibr.cs.tu-bs.de
Technische Universität Braunschweig
Braunschweig, Germany

Abstract

Symbolic execution is a dynamic software verification and testing technique that formally reasons about branches in a software under test using *satisfiability modulo theories (SMT)*. This requires encoding branch conditions as logical formulas in the SMT-LIB format consumed by SMT solvers. Due to the complex semantics of modern programming languages, this transformation is conducted based on an *intermediate representation (IR)*. To obtain this representation, prior work facilitates off-the-shelf compiler infrastructures (e.g., LLVM). Unfortunately, these established compilers are tailored to emitting program representations for fast execution on physical hardware, not for fast formal reasoning.

We propose integrating the compiler infrastructure with symbolic execution, thereby obtaining a program representation optimized for SMT solving. To explore this idea, we build upon an existing extensible IR to ease a co-design of the compiler, IR, and symbolic execution engine. We illustrate the potential of this integration by contributing *conditional representation*, a novel compiler transformation tailored to symbolic execution that results in a solver-efficient representation of pure functions with branches. Specifically, with conditional representation, such functions are represented as a single SMT expression, decreasing the number of solver queries while increasing per-query information. In our performed experiments, we symbolically execute real-world components from embedded operating systems and show that conditional representation reduces solver time by a geometric mean of 80 %.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; **Compilers**; **Interpreters**.

Keywords

Symbolic Execution, Intermediate Representation, SMT Encoding

ACM Reference Format:
Sören Tempel and Christian Dietrich. 2026. Towards a Vertically Integrated Compiler Infrastructure to Improve Solver Time during Symbolic Execution. In *Proceedings of the 8th International Workshop on Automated and verifiable Software sYstem DEvelopment (ASYDE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3843777.3844530

This work is licensed under a Creative Commons Attribution 4.0 International License.
ASYDE '26, Munich, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2989-8/2026/10
https://doi.org/10.1145/3843777.3844530

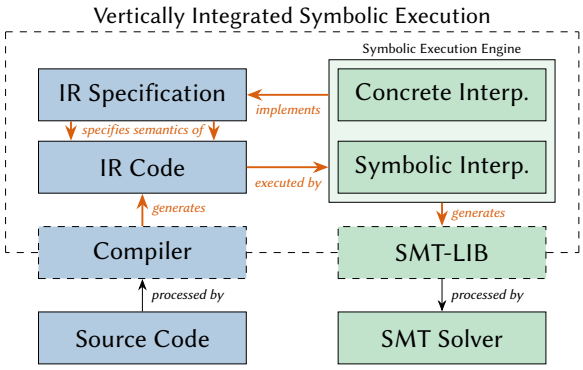


Figure 1: Overview of central components involved in source-based symbolic execution. Also illustrates components subject to vertical integration through our proposed approach.

1 Introduction

Symbolic execution [4] is an automated software testing and verification technique that is applicable to unmodified real-world programs. Therefore, it is well-suited to formally reason about the increasing number of third-party software artifacts and libraries that modern software is built upon [47]. However, a central challenge to this end is the limited scalability of symbolic execution [9].

Conceptually, symbolic execution formally reasons about branch points in the software under test using an SMT solver to determine satisfiability of branch conditions. As SMT solving is NP-hard, solver time is commonly cited as *the central bottleneck* for automated program verification and testing through symbolic execution [32, 27, 34]. The time required to solve a logical formula depends on its encoding in the standardized SMT-LIB [5] input format consumed by SMT solvers. As shown in Fig. 1, symbolic execution iteratively generates this SMT-LIB from an IR of the tested software, thus abstracting from the complex semantics of modern programming languages [37]. To obtain the IR, prior work facilitates off-the-shelf compiler infrastructures (e.g., LLVM [24]).

Unfortunately, established compilers optimize the IR for fast execution on real hardware, not for formal reasoning [46, 7, 18, 15, 50]. This is deeply problematic as execution on hardware and symbolic reasoning have fundamentally different optimization goals. For example, Zhang et al. illustrated that many off-the-shelf compiler optimizations (e.g., arithmetic strength reduction) are detrimental to solver time [50, Sect. 4.2]. Cadar thus suggested designing compiler transformations specifically for symbolic execution [7]. While prior work has proposed optimizing the SMT representation at runtime (e.g., via expression rewriting [10, 8, 34], state merging [23,

2, 41], or result reuse [48, 45, 27]), the design of novel compiler transformations for efficient SMT encoding remains understudied.

We argue that, instead of using off-the-shelf compilers and optimizations, the compiler infrastructure should be co-designed and integrated with symbolic execution. Thereby, enabling optimizations of SMT encoding at compile-time. Prior work on modular IRs, such as MLIR [25], has illustrated the potential of such domain-specific compilers outside the symbolic execution domain (e.g., for machine learning [1]). In this spirit, we build up the modular QBE IR [11] from prior work to vertically integrate and co-design the compiler, IR, and symbolic execution engine. We demonstrate the potential of this integration through our main contribution on *conditional representation*, a novel IR representation for pure functions, which reduces SMT solver time by a geometric mean of 80 %. To the best of our knowledge, this is the first proposal of a compiler optimization designed specifically for efficient SMT encoding.

Contributions. We claim the following technical contributions:

(1) We present Qute, a research vehicle built around the modular QBE IR that provides a symbolic executor and a framework to ease the creation of domain-specific program representations.
(2) Our main contribution is *conditional representation*, a novel program representation—tailored to symbolic execution—that achieves a more efficient SMT-LIB encoding for pure functions.

Outlook. In Sect. 2, we contrast the state-of-the-art with our proposed symbolic execution approach. We also present Qute, which we use as a research vehicle in Sect. 3 to implement conditional representation. In Sect. 4, we evaluate conditional representation by applying it to validate selected components of embedded operating systems. In Sect. 5, we argue that conditional representation preserves program semantics. Finally, we discuss related (Sect. 6) and future work (Sect. 7) and conclude the manuscript in Sect. 8.

2 Vertically Integrated Symbolic Execution

We provide a primer on the established source-based symbolic execution approach (Sect. 2.1) and then contrast this state-of-the-art with our approach by presenting the Qute framework (Sect. 2.2).

2.1 Source-based Symbolic Execution

Fig. 1 shows the components involved in standard source-based symbolic execution. The figure distinguishes the compiler side (blue) and the symbolic reasoning (green). The central component of this architecture is the *intermediate representation (IR)*. Symbolic execution is a dynamic software analysis technique and therefore executes the IR to explore the software under test based on defined *symbolic variables*. During execution, it iteratively tracks branches that depend on symbolic variables. Branch-by-branch, it performs a transformation to SMT-LIB and queries the solver for satisfiability of the condition and its negation, exploring all *reachable* branches.

In addition to the condition, all operations performed prior on involved symbolic variables (e.g., arithmetic operations) need to be represented in SMT-LIB. Therefore, a *symbolic interpreter* for the IR specification, which operates on SMT-LIB values, is required. Additionally, as symbolic reasoning is only performed for code interacting with symbolic variables, a *concrete interpreter* is also needed. As such, symbolic execution time is the composition of symbolic and concrete interpretation time. However, due to the involvement of SMT solving, the former is the common bottleneck [32, 27, 34].

2.2 Co-Design with the Compiler Infrastructure

Prior work reuses existing compiler infrastructures for the compiler side of symbolic execution (left-hand side of Fig. 1). Therefore, the compiler, its emitted IR code, and the IR specification are off-the-shelf components and not specifically tailored to symbolic execution. This design choice impacts solver time [46, 7, 18, 15, 50].

We contribute Qute, a vertically integrated symbolic execution stack in which the compiler, IR, and symbolic reasoning are tightly coupled to improve performance by enabling a co-design of these components (red path in Fig. 1). This idea is inspired by prior work on modular IRs (e.g., MLIR [25]), which eases extending the IR and enables the creation of domain-specific compilers. For symbolic execution, this facilitates the addition of high-level abstractions to the IR to model features available in source languages and SMT-LIB, but not in a conventional compiler IR (e.g., algebraic data types [5, pp. 60 sqq.]). Since the SMT-LIB is generated from the IR (see Fig. 1), semantic information for features not available in the IR is otherwise *lost in translation*. To ease adding custom features to an IR, the Qute framework is built around QBE [11], a modular IR from prior work that is explicitly designed to be easily modifiable [12].

In addition to an extensible IR (like MLIR [25] or QBE [11]), applying the outlined idea to symbolic execution also requires an *extensible symbolic execution engine*. That is, high-level abstractions added to the IR also need to be supported in the symbolic—and concrete—interpreter utilized by a symbolic execution engine (see Fig. 1). For this purpose, Qute provides a symbolic executor for the QBE IR that builds upon prior work on *modular interpreters* [42, 26, 6] to implement IR semantics in a way that abstracts from the specific value domain. This lets us derive both concrete and symbolic interpreters from the same abstraction and eases extending the IR with new operations and data types to improve its mapping to SMT-LIB. To utilize newly added IR features to represent a software under test, Qute further supports the creation of custom compiler passes. To this end, it reuses common static analysis techniques from prior work, for example, CFG-based dominator analysis [40] or control dependence graph generation [17]. Created compiler passes can be used in conjunction with existing compiler frontends for QBE. For instance, the cproc [19] C11 compiler, which we facilitate for our experimental evaluation in Sect. 4. On the symbolic execution side, Qute provides a standard implementation of dynamic symbolic execution [4, Sect. 2.1] with an address concretization [4, Sect. 3.2] memory model and a depth-first search path selection [4, Sect. 2.2]. In total, Qute is written in roughly 8000 SLOC in Haskell.

The implementation of the Qute framework is built upon established ideas from the symbolic execution, static analysis, modular IR, and interpreters domain. The result is a *research vehicle* that enables us to investigate the co-design of compiler and symbolic execution engine (see Fig. 1). By building upon a modular IR and modular interpreters, Qute allows us to co-design the IR specification, compiler passes, and symbolic executor to improve the semantic alignment with SMT-LIB. In the following, we present a novel compiler transformation that, through a co-design methodology, improves the SMT solver time by a geometric mean of 80 %.

```
char getsymbol(uint8_t code) {
  if (code <= BASE64_CAPITAL_UPPER_BOUND) { // C1
    return (code + 'A');
  }

  if (code <= BASE64_SMALL_UPPER_BOUND) { // C2
    return (code + ('z' - BASE64_SMALL_UPPER_BOUND));
  }

  if (code <= BASE64_NUMBER_UPPER_BOUND) { // C3
    return (code + ('9' - BASE64_NUMBER_UPPER_BOUND));
  }

  return (char)BASE64_NOT_DEFINED;
}
```

Figure 2: Excerpt of a Base64 encoding function from RIOT.

```
(define-fun getsymbol ((code (_ BitVec 8))) (_ BitVec 8)
  ;; C1: code <= BASE64_CAPITAL_UPPER_BOUND
  (ite (bvule code BASE64_CAPITAL_UPPER_BOUND)
       (bvadd code ASCII_A)
       ;; C2: code <= BASE64_SMALL_UPPER_BOUND
       (ite (bvule code BASE64_SMALL_UPPER_BOUND)
            (bvadd
              code
              (bvsub ASCII_z BASE64_SMALL_UPPER_BOUND))
            ;; C3: code <= BASE64_NUMBER_UPPER_BOUND
            (ite (bvule code BASE64_NUMBER_UPPER_BOUND)
                 (bvadd
                   code
                   (bvsub ASCII_9 BASE64_NUMBER_UPPER_BOUND))
                 BASE64_NOT_DEFINED))))
```

Figure 3: Representation of `getsymbol` in SMT-LIB.

3 Conditional Representation

We first expand on the interaction between the compiler, the IR, and the symbolic execution engine using a motivational example (Sect. 3.1). We then facilitate co-design through our Qute framework to optimize the representation of pure functions for symbolic execution (Sect. 3.2). Finally, we evaluate both symbolic (Sect. 4.1) and concrete (Sect. 4.2) execution time on a set of benchmarks.

3.1 Motivational Example

Consider the C code in Fig. 2, an excerpt of the `getsymbol` function from the Base64 [22] implementation of the RIOT [3] operating system.[1] For Base64 encoding, this function transforms a single input byte to its corresponding Base64 representation [22, Sect. 4]. If the parameter `code` is a symbolic variable, `getsymbol` needs to be transformed to an equivalent SMT-LIB representation. As described in Sect. 2.1, symbolic executors reason about the software under test branch-by-branch, and thus generate multiple SMT-LIB queries for `getsymbol`. The underlying SMT solver then considers satisfiability for one branch condition at a time. However, what if we supply the solver with more information at once: would that enable the SMT solver to reason about the `getsymbol` function more efficiently?

In our case, `getsymbol` is a pure function: its return value depends only on its symbolic `code` parameter. It does not interact with global variables, treats all local variables as immutable, and has no environment interactions (e.g., file system access). Solvers consume mathematical formulas in the SMT-LIB [5] language; in this language we can describe pure mathematical computations. Since it only performs such computations, the entirety of the `getsymbol` function can be represented as a single SMT-LIB expression. Doing so expands the information provided to the solver per query and—as we will see in Sect. 4.1—reduces overall solver time by decreasing the number of emitted SMT solver queries.

To express `getsymbol` as a single SMT-LIB expression, we need to capture the three conditional arithmetic operations on the symbolic `code` input. For this, we can use the *if-then-else (ITE)* function provided by SMT-LIB's Core theory. ITE takes a Boolean condition and two values, selecting the first value if the condition is true and the second otherwise [44].

Fig. 3 provides a complete SMT-LIB representation of `getsymbol`. The entirety of the C function is captured in a single SMT-LIB expression by nesting three ITE expressions, where each ITE covers one `if`-statement. The representation enables us to formally reason about the semantics of `getsymbol` using an SMT solver. For example, we can ask the solver for an assignment of the `code` parameter that causes `getsymbol` to return `BASE64_NOT_DEFINED`. Intuitively, this is an optimal representation as it contains the same semantic information as the original C code and thus provides the SMT solver all available information about the function in one query. Hence, reducing the number of generated queries.

Unfortunately, existing symbolic executors such as the popular KLEE [8] engine do not emit such expressions. KLEE facilitates the existing LLVM [24] compiler infrastructure to transform the C code to LLVM IR using the Clang C compiler and finally generates SMT-LIB from the IR. LLVM IR offers a `select` operation that is akin to the ITE expression from SMT-LIB and mapped accordingly by KLEE. However, using LLVM's `select` operations to represent all `if`-statements would imply that all (previously conditional) code would be eagerly evaluated. Therefore, choosing such a representation would be detrimental to execution time on real hardware. Since Clang targets real hardware, it abstains from representing `getsymbol` as a chain of LLVM `select` operations and instead uses conditional branches to represent the three `if`-statements. Hence, KLEE generates SMT-LIB from the LLVM IR branch-by-branch; emitting more solver queries and providing only a subset of all available information for the `getsymbol` function in each query.

In the following, we demonstrate how we can facilitate our Qute framework to closely integrate the symbolic execution with the compiler infrastructure to represent pure functions as a single SMT expression using SMT-LIB's ITE expression.

3.2 Optimized Intermediate Representation

The central challenge with respect to the motivational example is the utilization of branch instructions to represent a pure function. The symbolic IR interpreter cannot decide ahead of time if a basic block target of a conditional branch only performs pure arithmetic operations (i.e., can be represented in SMT-LIB). Therefore, it stops execution at the branch and determines satisfiability for the condition of this singular branch in isolation by querying the solver [8, Sect. 3]. Within the IR, we need to express that both targets of a conditional branch only perform pure arithmetic operations. If so, we can transform the involved basic blocks to single-path code without conditional branches (see Fig. 7), represent them as one SMT expression, and thus reduce the amount of SMT solver queries.

Prior work on *predicated execution* [33, 28, 38] in the computer architecture domain pursues a similar idea; they propose predicated instructions that are conditionally executed based on the state of

[1] The complete implementation is available in the RIOT Git repository.

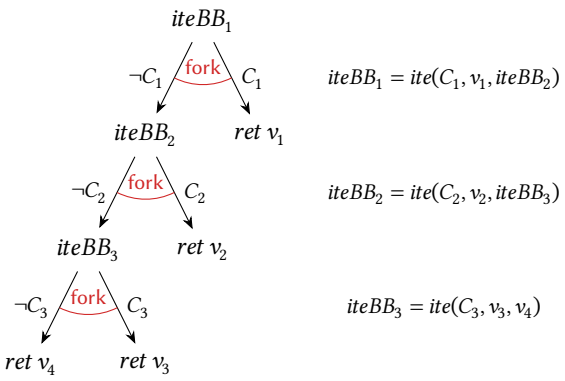


Figure 4: Conditional representation of the `getsymbol` function. Each node in the graph represents a basic block; the node text only shows the exit point instruction. Concrete execution forks the execution state at each `iteBB`, executing both children unconditionally. The right side represents the return value of each basic block. The value of the outermost basic block is the return value of the `getsymbol` function.

a predicate register. While this enables transformations to single-path code, predicated execution considers one instruction at a time ("if predicate is true, then execute the instruction"). This approach therefore only allows expressing if-then semantics, not if-then-*else*.

We facilitate our Qute framework to implement an idea that we refer to as *conditional representation*. Similar to predicated execution, conditional representation results in single-path code where all basic blocks are unconditionally executed. However, instead of operating on instruction granularity, we operate on the basic block level: at the exit point of a basic block—instead of performing a conditional branch—we select the "result" of one of two basic blocks based on a Boolean condition. This requires basic blocks to have at most two outgoing edges, which is always the case in the IR used by Qute. Further, it requires execution of involved basic blocks to not interfere with each other; hence, we restrict the functionality to *pure basic blocks* that are free of side effects.

For the implementation of conditional representation in Qute, we added a new exit point instruction to our extensible IR. We refer to this instruction as `iteBB`, it takes three parameters: one Boolean condition and two labels of pure basic blocks. Semantically, it *unconditionally executes both basic blocks* and yields the result of the first basic block if the condition is true or the result of the second otherwise. To obtain "the result of a basic block", it is executed until the first exit point that triggers a function return, the return value is the basic block result. Within a pure function, the outermost `iteBB` instruction signifies the return value of that function. Fig. 4 illustrates use of these exit point instructions for the representation of `getsymbol` and shows how `iteBB` iteratively composes nested ITE expressions for the desired SMT-LIB output (see Fig. 3).

To implement support for `iteBB` in Qute, we had to modify roughly 100 SLOC across 8 files. This also covers the changes required to both our concrete and symbolic interpretation and serves to illustrate that our framework enables domain-specific IR enhancement with minimal effort.

3.3 Compiler Optimization

Apart from adding an additional feature to the extensible IR used by our Qute framework, we also need to ensure that this feature is facilitated by the compiler. For example, for the C code from Fig. 2, we need to ensure that the compiler frontend uses the new `iteBB` instruction as illustrated in Fig. 4. For this purpose, we propose a compiler optimization for conditional representation.

Presently, our proposed optimization operates on function granularity, it optimizes a given function in the IR if it satisfies the following criteria:

(1) *Purity.* The function must be pure: its output must only depend on input function parameters. Further, it must treat all local variables as immutable.
(2) *Absence of Loops.* The function must not contain loops. Otherwise, due to the aforementioned semantics of our `iteBB` instruction, execution would never terminate.
(3) *Leaf Function.* The function must be a leaf in the *control flow graph (CFG)*. That is, it cannot call other functions and cannot, for example, trigger system calls.

Some of these limitations may be lifted in future work. For example, we could take loop bounds into consideration to ensure termination in the presence of loops (see Sect. 7). Our proposed compiler optimization also requires the IR to only assign variables once. This ensures that a basic block does not overwrite a variable originating elsewhere, which would constitute a non-pure side effect. Algorithms for converting an IR program to a static single-assignment form are well-described in the existing literature [17].

Using our Qute framework, we created a prototype implementation of the outlined compiler pass. This prototype does not optimize functions with store instructions and function calls (criterion 1. and 3.). Further, it excludes functions with loops through a CFG-based dominator analysis (criterion 2.) [40]. Functions that satisfy all outlined criteria only consist of pure basic blocks as defined in Sect. 3.2. For such functions, our prototype optimizer iterates over all basic blocks and replaces the conditional branch instructions at exit points with our newly added `iteBB` instruction. The described criteria ensure that the basic blocks do not interfere with each other and can thus both be executed sequentially (see Sect. 5).

The compiler pass benefits from established optimizations described in the existing compiler literature. For example, function inlining enables the conversion of additional functions to leafs in the CFG [14, 16]. Similarly, stack elimination techniques—which get rid of the function call stack—make more functions suitable for conditional representation by removing store instructions from the function's basic blocks [49].

The compiler pass implementation consists of approximately 100 SLOC in Haskell. The implementation is significantly simplified through the addition of the `iteBB` exit point instruction to our IR.

4 Evaluation

In the following, we evaluate our conditional representation compiler pass on a set of benchmark applications. As explained in Sect. 2.1, we distinguish concrete and symbolic execution time during symbolic program analysis. Therefore, we assess both the impact on symbolic (i.e., solver) time (Sect. 4.1) and concrete execution time (Sect. 4.2) using our benchmarks.

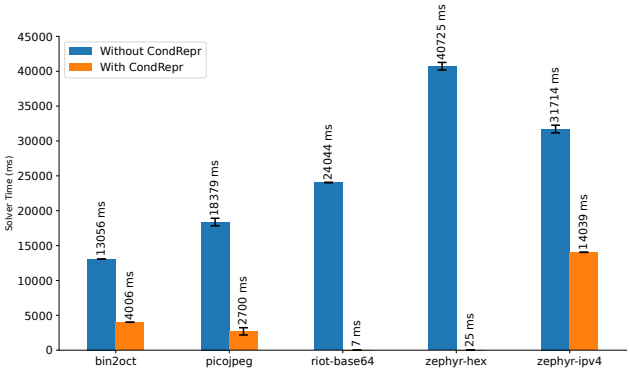


Figure 5: Solver time results for our benchmarks.

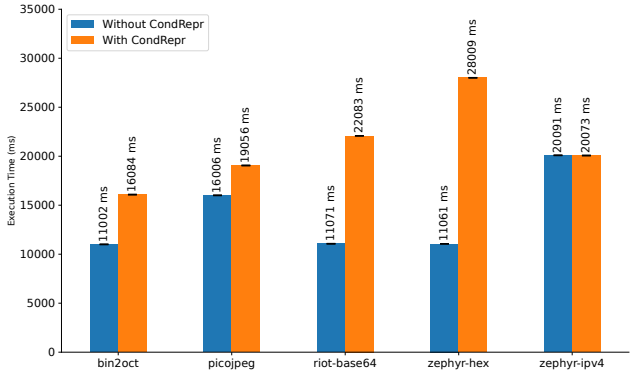


Figure 6: Execution time results for our benchmarks.

The benchmark suite we use consists of real-world code for embedded systems. This includes a JPEG decoder [20] and components taken from the embedded operating systems Zephyr and RIOT [3]. Further, we include a generic binary-to-octal number conversion routine from GitHub. We only selected components that contain functions satisfying our optimization criteria. We consider this a preliminary evaluation that we aim to expand upon in future work.

As a baseline for our experiments, we use the original version of Qute as described in Sect. 2.2 (i.e., without the conditional representation optimization). We do not compare against other symbolic execution engines as part of this preliminary evaluation, since isolating the effect of a single optimization from other implementation choices is known to be notoriously difficult [37, Sect. 5.1].

The data we present in the following represents an arithmetic mean over 3 executions. All experiments have been conducted on a Linux system with an Intel Ultra 9 processor and 94 GiB of memory.

4.1 Symbolic Execution Time

The conditional representation optimization is tailored to reduce the time spent on solving SMT constraints. Solver time is *the central bottleneck* of symbolic execution and thus a vital optimization criterion to improve its scalability [32, 27, 34]. To assess the impact of our optimization on solver time, we perform complete exhaustive exploration of all possible execution states within each of our benchmark applications. Further, we instructed Qute's symbolic executor to write all SMT constraints to a dedicated SMT-LIB file.

We then process the entire file 3 times using the Bitwuzla [31] SMT solver and report an arithmetic mean over all 3 solver invocations. We use Bitwuzla, since—in the 2025 SMT-COMP—it was one of the fastest solver for the bitvector logic facilitated by Qute.[2] The results are presented in Fig. 5 and Table 1. In the figure, we compare solver time results for Qute with- and without our conditional representation optimization. In total, utilization of our conditional representation compiler pass reduces solver time by a geometric mean of 80 %. On the `riot-base64` benchmark, utilization of conditional representation even decreases solver time by 99.97 %. This benchmark validates the Base64 [22] implementation of the RIOT [3] operating system from Sect. 3.1. Specifically, it asserts that `base64_decode` is the inverse of `base64_encode` and vice versa. Internally, RIOT's Base64 implementation uses two utility functions (`getsymbol` and `getcode`), which are responsible for en- and decoding a single character. Both of these functions are pure and thus represented as an ITE chain in SMT-LIB using our compiler pass (refer to Fig. 3 for the representation of `getsymbol`). As per Table 1, doing so reduces the amounts of queries generated—and thus solver time—for the inversion property, from 319,530 queries without conditional representation to 16 with it enabled. Similarly, on the `zephyr-hex` benchmark, we also observe a speedup of 99.94 %. In this benchmark we again represent the entirety of two decoding functions as an ITE chain in SMT-LIB (`char2hex` and `hex2char`),[3] reducing the amount of generated queries from 427,552 to 271. As shown in Table 1, conditional representation achieves this reduction by increasing the per-query complexity, which we approximate here through the average SMT-LIB expression depth.

However, we also observe a significant speedup for benchmarks where we do not represent entire encoding/decoding routines in SMT-LIB. For example, for the `zephyr-ipv4` address parser benchmark we only optimize four utility functions from the standard library (`isupper`, `isalpha`, `isspace`, and `isdigit`). Nonetheless, representing these small functions using ITE in SMT-LIB results in a solver time reduction of 55.73 % as they are used frequently in the codebase. Consequentially, these preliminary results illustrate the potential of our conditional representation optimization wrt. boosting solver time for symbolic execution of real-world code.

4.2 Concrete Execution Time

As discussed in Sect. 2.1, overall symbolic execution performance is influenced by both the symbolic (i.e., solver) and concrete execution time. The latter is performed when executed code does not interact with symbolic variables. Therefore, apart from solver time—which our conditional representation is designed to reduce—we also need to assess the impact on concrete execution time. To this end, we modify the benchmark applications to repeatably process a fixed set of concrete inputs, ensuring execution through Qute's concrete interpreter. We execute the modified benchmarks 3 times using concrete interpretation with and without conditional representation and present the arithmetic mean over these executions in Fig. 6.

[2]For more information refer to https://smt-comp.github.io/2025/results/.

[3]Refer to the `lib/utils/hex.c` file in the Zephyr Git Repository.

Table 1: Amount of generated SMT solver queries and average nesting depth across all generated SMT-LIB expressions.

	Without CondRepr.		With CondRepr.	
Bench	#Queries	Avg. Depth	#Queries	Avg. Depth
bin2oct	2,265	40	78	21
picojpeg	51,041	66	7,177	121
riot-base64	319,530	13	16	27
zephyr-hex	427,552	10	271	28
zephyr-ipv4	61,357	7	29,507	8

Naturally, our conditional representation optimization causes an increase in execution time since optimized functions are now single-path code without conditional branches (i.e., more code is unconditionally executed). This illustrates why off-the-shelf compiler infrastructures (e.g., LLVM [24]), which optimize for fast concrete execution, do not choose this IR representation. In total, conditional representation increases concrete execution time by a less significant geometric mean of 16 %. For the `zephyr-ipv4` benchmark, where we only optimized four utility functions, we do not observe a significant difference. We attribute this to the fact that—since these functions are very small—only 2-3 instructions are executed more per invocation with our conditional representation transformation.

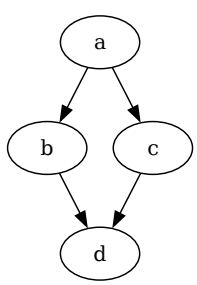
5 Preservation of Program Semantics

Symbolic execution is generally a sound software analysis technique and not subject to false positives [4]. To uphold this property, it is essential to ensure that our conditional representation transformation preserves the program semantics. Otherwise, the symbolic executor would reason about a different program, and discovered defects may not reproduce in the original. This can yield false positives and void the soundness of the symbolic analysis. In the following, we thus justify that the transformation is semantics-preserving.

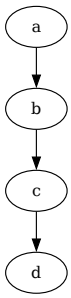
Similar to prior work on predicated execution [33, 28, 38], conditional representation conceptually transforms code with conditional branches to single-path code with unconditional branches. Consequentially, all instructions in an optimized pure function are unconditionally executed. Fig. 7 visualizes this transformation: Without the optimization (Fig. 7a), only basic block *b xor c* is executed on a given path; with conditional representation, both are executed sequentially (Fig. 7b). This transformation is only legal if the involved basic blocks do not interfere with each other. For example, if *b* modifies a global variable that is later read by *c*, then the transformation would not be semantic-preserving. Therefore, as per Sect. 3.3, the compiler pass only optimizes pure functions that treat *all* variables as immutable (*purity*). Similarly, it only targets *leaf functions* to prevent side effects through functions and system calls. This ensures that the—previously conditionally executed—basic blocks do not interfere with each other and can be sequentially executed in arbitrary order while preserving semantics.

A further challenge wrt. single-path code is loop handling [38, Sect. 3]. If we eliminate conditional branches, loops would never terminate. Our prototype sidesteps this problem by not optimizing functions with loops that have not been unrolled. Recursive functions are also not optimized as they are not leafs in the CFG.

We leave a full formal correctness proof for future work. However, *for all functions optimized in our benchmarks* (Sect. 4), we have verified that transformed functions preserve semantics using equivalence checking with exhaustive symbolic execution [8, Sect. 5.5]. That is, for a conditional function f, which is transformed via conditional representation to f', we have added a test harness: $f(x) \equiv f'(x)$. We then marked all function arguments x as symbolic variables and exhaustively explored the state space using Qute's symbolic executor. For all functions optimized in our benchmark suite, this property holds, proving that the transformation is semantic-preserving on all applications from our benchmark suite.



(a) CFG before the optimization.



(b) CFG after the optimization.

Figure 7: Conditional representation for if-statements.

6 Related Work

We discuss related work studying optimizations for efficient SMT encoding and the influence of compilers on symbolic execution.

6.1 Optimizing SMT Encoding

Solver time is commonly cited as the central bottleneck of symbolic execution [32, 27, 34]. As solver time is influenced by the SMT representation passed to the solver, a large body of prior work concerns itself with optimizing this representation. Prior work on EXE [10] and KLEE [8, 34] presented several expression rewriting optimizations, which are performed on SMT-LIB expressions before the solver is invoked. Similarly, prior work on state merging [23, 2, 41] fuses different paths into a single path by obtaining a disjunction of their respective path constraints. This is closely related to our work, as it is subject to the same trade-off: decreasing the number of paths increases the per-query complexity. In contrast to our work, the aforementioned optimizations are applied dynamically during the symbolic exploration. Therefore, for state merging, whether to merge states is decided based on heuristics at runtime; these must be fast and cannot require costly analysis [4, Sect. 5.6]. A compile-time approach like ours can take existing compiler information into account and thus make a more informed decision faster. For example, in our work, we use the dominator analysis results available within a compiler context to select optimization candidates [40]. In this lineage, prior work outside the symbolic execution domain has already illustrated that compiler analysis results are beneficial for SMT encoding [30]. Lastly, there is related work in the *bounded model checking (BMC)* domain, which generates SMT-LIB from an IR [29, 21, 13]. For example, SeaHorn includes a more rudimentary ITE chain optimization that only transforms

switch statements [21, Sect. 2]. Generally, in contrast to symbolic execution, BMC does not dynamically execute the tested software and performs a static transformation to SMT-LIB. As conditional representation builds the ITE chains dynamically during execution (see Fig. 4), it can transform more than just switch statements but is as-is not applicable to BMC. Further, BMC is thus not subject to the conflicting optimization goals of concrete and symbolic execution (Sect. 4), which—for solver time optimizations—have not been reported in the existing symbolic execution compiler literature. Thus, we cannot apply BMC optimizations directly to symbolic execution.

6.2 Influence of Compilers

Prior work has investigated the interaction between compiler, intermediate representation, and symbolic execution. Specifically, Poeplau et al. [37] have studied the influence of different IR generation approaches on symbolic execution. They focus on the process by which the IR representation is obtained and compare its generation from both source and binary code. Among other things, they observed that binary-based execution leads to "harder SMT queries" and discuss compiler optimizations as a potential reason [37, Sect. 6.2]. In this spirit, related work has empirically quantified the impact of established compiler optimizations on symbolic execution. This work is closely related to ours, as it illustrates that optimizations beneficial for concrete execution slow down symbolic execution [46, 18, 15, 50]. As a novel insight, we provide preliminary data that also shows the inverse: optimizations beneficial for symbolic execution can slow down concrete execution. Beyond established passes, prior work motivated the creation of compiler transformation tailored to symbolic execution but does not propose specific transformations [7]. Further, prior work has also proposed compilation-based symbolic execution [35, 36, 39], which boosts concrete execution time through native (i.e., compiled) execution of the tested code. However, to the best of our knowledge, our conditional representation optimization is the first compiler pass designed to reduce the vital SMT-solver time of symbolic execution.

7 Future Work

The central challenge wrt. the general idea of integrating symbolic execution and compiler infrastructures is the conflicting optimization goals of concrete and symbolic execution. Where prior work has shown that optimizations beneficial for concrete execution slow down symbolic execution [46, 18, 15, 50], we provide preliminary data indicating that the inverse also holds. Since overall symbolic execution time is the composition of both, it is vital to overcome this dilemma. An interesting direction for future work would be to investigate optimizing the software under test twice: once for concrete and once for symbolic execution. This, however, creates a challenge wrt. ensuring execution equivalence.

Further, we see opportunities to improve our prototype implementation of conditional representation. Specifically, we aim to make it applicable to more code by analyzing individual basic blocks instead of entire functions. Congruent with prior work [23, 2, 41], a challenge to this end is that conditional representation decreases the number of SMT queries by increasing the per-query complexity. Utilizing single-path conditional representation for more code may result in SMT queries that take too long to solve. We aim to assess this aspect in future work by expanding our benchmarks. Lastly, we see potential to develop additional code transformations for symbolic execution using our Qute framework [7]. For this, we aim to identify SMT-LIB language features that are especially beneficial for solver time and then optimize the compiler infrastructure to retain this information. A promising direction is a more faithful representation of data types in the IR. For example, enabling a mapping to SMT-LIB's algebraic datatypes [5, pp. 60 sqq.].

8 Conclusion

Symbolic execution formally reasons about a software under test by representing its semantics in a unifying IR. As this presupposes the existence of a compiler that emits a representation in this IR, existing symbolic execution work builds upon off-the-shelf compiler infrastructures (e.g., LLVM [24]). These established compiler infrastructures are tailored to emitting program representations optimized for fast execution on physical hardware, not for fast formal reasoning using an SMT solver. In this paper, we present the Qute framework, which enables a vertical integration of the compiler, intermediate representation, and the symbolic execution engine. We demonstrate its capabilities through a case study in which we contribute a novel program transformation that we refer to as conditional representation. The transformation enables the representation of pure functions as singular SMT expressions. Our experiments show that conditional representation reduces SMT solver time (the central bottleneck of symbolic execution) by a geometric mean of 80 %, while preserving program semantics. However, it also increases concrete execution time by a less significant margin of 16 %. For future work, we aim to mitigate the well-known conflicting optimization goals of concrete and symbolic execution.

Data Availability

Qute is available as an open-source Haskell library.[4] All artifacts required to reproduce the experiments are available via Zenodo [43].

Acknowledgments

This work was partly funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 501887536.

References

[1] Jason Ansel et al. 2024. PyTorch 2: faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (ASPLOS '24). Association for Computing Machinery, New York, USA, 929–947. doi:10.1145/3620665.3640366.

[2] Thanassis Avgerinos et al. 2014. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering* (ICSE). Association for Computing Machinery, Hyderabad, India, 1083–1094. ISBN: 9781450327565. doi:10.1145/2568225.2568293.

[3] Emmanuel Baccelli et al. 2018. RIOT: an open source operating system for low-end embedded devices in the IoT. *IEEE Internet of Things Journal*. IoT-J 5, 6, (Dec. 2018), 4428–4440. doi:10.1109/JIOT.2018.2815038.

[4] Roberto Baldoni et al. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51, 3, (May 2018). doi:10.1145/3182657.

[5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2021. The SMT-LIB Standard Version 2.6. Standard. (May 12, 2021). https://smt-lib.org/papers/smt-lib-reference-v2.6-r2021-05-12.pdf.

[6] Thomas Bourgeat et al. 2023. Flexible instruction-set semantics via abstract monads (experience report). *Proceedings of the ACM on Programming Languages*, 7, ICFP, (Aug. 2023). doi:10.1145/3607833.

[4] https://hackage.haskell.org/package/qute

[7] Cristian Cadar. 2015. Targeted program transformations for symbolic execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (ESEC/FSE). Association for Computing Machinery, Bergamo, Italy. ISBN: 9781450336758. doi:10.1145/2786805.2803205.

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (OSDI). USENIX Association, San Diego, California, 209–224.

[9] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56, 2, (Feb. 2013), 82–90. doi:10.1145/2408776.2408795.

[10] Cristian Cadar et al. 2008. EXE: automatically generating inputs of death. *ACM Transactions on Privacy and Security*. TOPS 12, 2, (Dec. 2008). doi:10.1145/1455518.1455522.

[11] Quentin Carbonneaux. 2023. QBE intermediate language. Retrieved May 22, 2026 from https://c9x.me/compile/doc/il-v1.2.html.

[12] Quentin Carbonneaux. 2021. QBE vs LLVM. Retrieved May 22, 2026 from https://c9x.me/compile/doc/llvm.html.

[13] Montgomery Carter et al. 2016. Smack software verification toolchain. In *Proceedings of the 38th International Conference on Software Engineering Companion* (ICSE). Association for Computing Machinery, Austin, Texas, 589–592. ISBN: 9781450342056. doi:10.1145/2889160.2889163.

[14] P. P. Chang and W.-W. Hwu. 1989. Inline function expansion for compiling C programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation* (PLDI). Association for Computing Machinery, Portland, USA, 246–257. ISBN: 089791306X. doi:10.1145/73141.74840.

[15] Junjie Chen et al. 2018. Learning to accelerate symbolic execution via code transformation. In *32nd European Conference on Object-Oriented Programming* (ECCOP). Todd Millstein, (Ed.) Vol. 109. Dagstuhl, Germany, 6:1–6:27. ISBN: 9783959770798. doi:10.4230/LIPIcs.ECOOP.2018.6.

[16] Keith D. Cooper, Mary W. Hall, and Linda Torczon. 1991. An experiment with inline substitution. *Software: Practice and Experience*, 21, 6, 581–601. doi:10.1002/spe.4380210604.

[17] Ron Cytron et al. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*. TOPLAS 13, 4, (Oct. 1991), 451–490. doi:10.1145/115372.115320.

[18] Shiyu Dong et al. 2015. Studying the influence of standard compiler optimizations on symbolic execution. In *2015 IEEE 26th International Symposium on Software Reliability Engineering* (ISSRE), 205–215. doi:10.1109/ISSRE.2015.7381814.

[19] Michael Forney. 2025. cproc. Version 70511143. SourceHut. (Nov. 19, 2025). Retrieved Jan. 7, 2026 from https://git.sr.ht/~mcf/cproc.

[20] Rich Geldreich. 2020. picojpeg. Version 8ab33a. GitHub. (Mar. 23, 2020). Retrieved May 23, 2026 from https://github.com/richgel999/picojpeg.

[21] Arie Gurfinkel et al. 2015. The SeaHorn verification framework. In *Computer Aided Verification* (CAV). Daniel Kroening and Corina S. Păsăreanu, (Eds.) Springer International Publishing, Cham, 343–361. ISBN: 9783319216904. doi:10.1007/978-3-319-21690-4_20.

[22] Simon Josefsson. 2006. The Base16, Base32, and Base64 data encodings. RFC 4648. (Oct. 2006). doi:10.17487/RFC4648.

[23] Volodymyr Kuznetsov et al. 2012. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI). Association for Computing Machinery, Beijing, China, 193–204. ISBN: 9781450312059. doi:10.1145/2254064.2254088.

[24] Chris Lattner and Vikram Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization* (CGO). San Jose, California, (Mar. 2004), 75–86. doi:10.1109/CGO.2004.1281665.

[25] Chris Lattner et al. 2021. MLIR: scaling compiler infrastructure for domain specific computation. In *International Symposium on Code Generation and Optimization* (CGO), 2–14. doi:10.1109/CGO51591.2021.9370308.

[26] Sheng Liang and Paul Hudak. 1996. Modular denotational semantics for compiler construction. In *Programming Languages and Systems* (ESOP). Hanne Riis Nielson, (Ed.) Springer Berlin Heidelberg, Berlin, Heidelberg, 219–234. ISBN: 9783540499428. doi:10.1007/3-540-61055-3_39.

[27] Tianhai Liu et al. 2014. A comparative study of incremental constraint solving approaches in symbolic execution. In *Hardware and Software: Verification and Testing* (HVC). Eran Yahav, (Ed.) Springer International Publishing, 284–299. ISBN: 9783319133386. doi:10.1007/978-3-319-13338-6_21.

[28] S.A. Mahlke et al. 1992. Effective compiler support for predicated execution using the hyperblock. In *Proceedings the 25th Annual International Symposium on Microarchitecture* (MICRO), 45–54. doi:10.1109/MICRO.1992.696999.

[29] Florian Merz, Stephan Falke, and Carsten Sinz. 2012. LLBMC: bounded model checking of C and C++ programs using a compiler IR. In *Verified Software: Theories, Tools, Experiments* (VSTTE). Rajeev Joshi, Peter Müller, and Andreas Podelski, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, 146–161. ISBN: 9783642277054. doi:10.1007/978-3-642-27705-4_12.

[30] Benjamin Mikek and Qirun Zhang. 2023. Speeding up SMT solving via compiler optimization. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ESEC/FSE). Association for Computing Machinery, San Francisco, CA, USA, 1177–1189. ISBN: 9798400703270. doi:10.1145/3611643.3616357.

[31] Aina Niemetz and Mathias Preiner. 2023. Bitwuzla. In *Computer Aided Verification* (CAV). Constantin Enea and Akash Lal, (Eds.) Springer Nature Switzerland, 3–17. ISBN: 9783031377037. doi:10.1007/978-3-031-37703-7_1.

[32] Hristina Palikareva and Cristian Cadar. 2013. Multi-solver support in symbolic execution. In *Computer Aided Verification* (CAV). Natasha Sharygina and Helmut Veith, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, 53–68. ISBN: 9783642397998. doi:10.1007/978-3-642-39799-8_3.

[33] Jospeh C. H. Park and Mike Schlansker. 1991. On Predicated Execution. Tech. rep. HPL-91-58. Hewlett Packard, (May 1991).

[34] David M. Perry et al. 2017. Accelerating array constraints in symbolic execution. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA). Association for Computing Machinery, Santa Barbara, CA, USA, 68–78. doi:10.1145/3092703.3092728.

[35] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, (Aug. 2020), 181–198. ISBN: 9781939133175.

[36] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: compilation-based symbolic execution for binaries. In *The Network and Distributed System Security Symposium 2021* (NDSS). (Feb. 2021). doi:10.14722/NDSS.2021.24118.

[37] Sebastian Poeplau and Aurélien Francillon. 2019. Systematic comparison of symbolic execution systems: intermediate representation and its generation. In *Proceedings of the 35th Annual Computer Security Applications Conference* (ACSAC), 163–176. ISBN: 9781450376280. doi:10.1145/3359789.3359796.

[38] Daniel Prokesch, Stefan Hepp, and Peter Puschner. 2015. A generator for time-predictable code. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing* (ISORC), 27–34. doi:10.1109/ISORC.2015.40.

[39] Zhenxiao Qi et al. 2024. SymFit: making the common (concrete) case fast for binary-code concolic execution. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, (Aug. 2024), 415–432. ISBN: 9781939133441.

[40] G. Ramalingam. 2002. On loops, dominators, and dominance frontiers. *ACM Transactions on Programming Languages and Systems*. TOPLAS 24, (Sept. 2002), 5, (Sept. 2002). doi:10.1145/570886.570887.

[41] Koushik Sen et al. 2015. MultiSE: multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (ESEC/FSE). Association for Computing Machinery, Bergamo, Italy, 842–853. ISBN: 9781450336758. doi:10.1145/2786805.2786830.

[42] Guy L. Steele. 1994. Building interpreters by composing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL). Association for Computing Machinery, Portland, Oregon, USA, 472–492. ISBN: 0897916360. doi:10.1145/174675.178068.

[43] Sören Tempel. Artifacts for ASYDE'26: towards a vertically integrated compiler infrastructure to improve solver time during symbolic execution. Zenodo, (Aug. 2026). doi:10.5281/zenodo.22095861.

[44] Cesare Tinelli. 2024. The Satisfiability Modulo Theories Library. Core Theory. Standard. Version 2.7. (July 21, 2024). https://smt-lib.org/theories-Core.shtml.

[45] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (FSE). Association for Computing Machinery, Cary, North Carolina. ISBN: 9781450316149. doi:10.1145/2393596.2393665.

[46] Jonas Wagner, Volodymyr Kuznetsov, and George Candea. 2013. -OVERIFY: optimizing programs for fast Verification. In *14th Workshop on Hot Topics in Operating Systems* (HotOS). USENIX Association, (May 2013).

[47] Laurie Williams et al. 2025. Research directions in software supply chain security. *ACM Transactions on Software Engineering and Methodology*. TOSEM 34, 5, (May 2025). doi:10.1145/3714464.

[48] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. 2012. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (ISSTA). Association for Computing Machinery, Minneapolis, USA, 144–154. ISBN: 9781450314541. doi:10.1145/2338965.2336771.

[49] Xuejun Yang, Nathan Cooprider, and John Regehr. 2009. Eliminating the call stack to save RAM. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (LCTES). Association for Computing Machinery, Dublin, Ireland, 60–69. ISBN: 9781605583563. doi:10.1145/1542452.1542461.

[50] Yue Zhang et al. 2024. When compiler optimizations meet symbolic execution: an empirical study. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (CCS). Association for Computing Machinery, 4212–4225. ISBN: 9798400706363. doi:10.1145/3658644.3670372.

Received 2026-07-17; accepted 2026-08-23