Compiling Parallel Symbolic Execution with Continuations

Guannan Wei*, Songlin Jia*, Ruiqi Gao*, Haotian Deng*, Shangyin Tan†, Oliver Bračevac*, Tiark Rompf*
*Department of Computer Science, Purdue University
West Lafayette, IN, USA
{guannanwei,jia137,gao606,deng254,bracevac,tiark}@purdue.edu
†Department of EECS, UC Berkeley
Berkeley, CA, USA
shangyin@berkeley.edu

***Abstract*—Symbolic execution is a powerful program analysis and testing technique. Symbolic execution engines are usually implemented as interpreters, and the induced interpretation overhead can dramatically inhibit performance. Alternatively, implementation choices based on instrumentation provide a limited ability to transform programs. However, the use of compilation and code generation techniques beyond simple instrumentation remains underexplored for engine construction, leaving potential performance gains untapped.**

In this paper, we show how to tap some of these gains using sophisticated compilation techniques: We present GENSYM, an optimizing symbolic-execution compiler that generates symbolic code which explores paths and generates tests in parallel. The key insight of GENSYM is to compile symbolic execution tasks into cooperative concurrency via continuation-passing style, which further enables efficient parallelism. The design and implementation of GENSYM is based on partial evaluation and generative programming techniques, which make it high-level and performant at the same time. We compare the performance of GENSYM against the prior symbolic-execution compiler LLSC and the state-of-the-art symbolic interpreter KLEE. The results show an average 4.6× speedup for sequential execution and 9.4× speedup for parallel execution on 20 benchmark programs.

***Index Terms*—symbolic execution, compiler, code generation, metaprogramming, continuation**

I. INTRODUCTION

Symbolic execution (SE for short) [1]–[4] is a popular program analysis technique which is effective at test-suite generation and bug finding. The key idea is to use symbolic values to represent unknown runtime inputs of programs. By collecting path constraints imposed on symbolic values, we obtain logical constraints describing inputs that trigger certain program behaviors. Modern SE was made practical by improvements in constraint solving (*e.g.*, SMT [5]) and techniques to combine concrete and symbolic execution, such as execution-generated testing [6] and concolic execution [7], [8]. However, SE still faces challenges at scale, such as path explosion and environment modeling issues [9], [10].

A well-studied avenue to improve the performance of SE is semantics-preserving program transformations [11]–[13] applied before passing programs to an SE *engine*. However, the execution mechanism behind most SE engines leaves further optimization opportunities untapped. They work as symbolic interpreters of source programs, in contrast to how performance-critical tools are commonly implemented, *i.e.*, as compilers. This practice misses opportunities to further improve the performance of SE and advance the development of engines.

In this paper, we study constructing scalable SE engines using code generation and compiler techniques. We present the design, implementation, and evaluation of GENSYM, a symbolic-execution compiler for the LLVM intermediate representation (IR). Given an input LLVM IR program, GENSYM generates C++ code that schedules parallel path exploration and orchestrates SMT solver invocations. Running this C++ program generates test cases for explored paths and failed assertions. The novel contribution of GENSYM is its aggressive use of code generation techniques for a *systematic*, *principled*, and *high-level* construction process yielding *optimized* and *performant* symbolic execution engines.

From Interpreters to Compilers. Interpreters are an easy and high-level approach for building symbolic execution engines. However, a naively built interpreter without carefully engineered optimizations can be orders of magnitude slower than a compiled program. One of the major contributors to the slowdown is the interpretation overhead [14] from inspecting and dispatching on the program representation. This overhead is absent in compiled programs.

Nevertheless, building a compiler from scratch is difficult, error prone, and perhaps not economic in many cases. The fact that SE induces a non-standard language semantics adds to the challenges of compiler construction from scratch.

We adopt recent approaches developed for compiled domain-specific languages (DSLs) [15]–[19], which greatly simplify developing SE compilers by making use of higher-level programming abstractions [20]. The language with its symbolic semantics is understood as a DSL, which can be realized as a symbolic interpreter. To compile this DSL, we exploit Futamura's observation from the 1970s [21], [22] which states that interpretation and compilation are fundamentally connected. In essence, GENSYM is about partially evaluating [14] (*i.e.*, specializing) a symbolic interpreter given an input program. The result of partial evaluation is a faster program that is equivalent to running the input program under the interpreter. Therefore, partially evaluating interpreters is equivalent to compilation (a.k.a. the first Futamura projection).

Building on this foundation and modern code generation techniques, the development effort for the GENSYM compiler is as lightweight as developing an ordinary high-level

† Work done at Purdue University.

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00116

(symbolic) interpreter for a programming language. At the same time, GENSYM exhibits the efficiency and performance of compilation, since it completely eliminates interpretation overhead in the generated code: there is no residual syntactic component (*e.g.*, AST or IR) of source programs left at runtime, and inspection/dispatch on source program representations is entirely absent.

Beyond Concolic Semantics and Instrumentation. Although several recent works (*e.g.*, [23], [24]) have advocated compilation-based symbolic execution, they compile *concolic semantics*, where only a single symbolic path is guided by concrete inputs. Moreover, the approach to compilation in these works is using commodity compiler frameworks (*e.g.*, LLVM) to *instrument* code at the IR level, and then generate executables with existing compilation pipelines.

This "instrumentation-by-compilation" approach works well for single-path concolic execution, but cannot effectively scale to multipath symbolic execution with sophisticated search strategies, which requires generating non-trivial code that explores multiple paths with no interference. The key challenge is to represent program states (*e.g.*, the heap) and control structures (*e.g.*, branching) so that we can manipulate and transform them conforming to some multi-path symbolic semantics (*e.g.*, pausing/resuming execution of paths).

GENSYM tackles this challenge by generating code in continuation-passing style (CPS) [25], a widely used compiler IR in functional languages [26]–[30]. CPS represents control flow explicitly as first-class entities that can be stored in data structures, permitting suspension and resumption of path execution. After a path is executed for a while, its execution proactively yields the control to a scheduler, which selects the next interesting path to explore (*e.g.*, guided by a heuristic). This is essentially a form of *cooperative concurrency*.

Scheduling Parallel Execution. The nature of symbolic semantics makes it suitable for parallelization, since little communication is needed between paths. However, running interpreters in parallel is not optimal, since the interpretation overhead is observed repeatedly for each interpreter instance.

Compiling to cooperative multitasking with continuations not only eliminates the interpretation overhead, but also unlocks parallelism using multicore CPUs (Section IV) with little memory and context-switching overhead. Once the scheduler regains control, it dispatches paths to workers that run in parallel from a fixed-sized thread pool of OS threads.

Prior work on LLSC [31] also generates parallel SE code. Without using CPS, it however lacks fine-grained manipulation of control structures, and thus uses C++'s asynchrony primitives with limited scalability. To the best of our knowledge, GENSYM is the first tool that achieves performant parallel SE using continuations and cooperative concurrency.

Generating Optimized Code. Previous works have shown that compiler optimizations over source programs (*e.g.*, CFG simplification) can significantly alter the performance of symbolic execution over these programs [11]–[13].

Orthogonal to source code optimizations, GENSYM generates code optimized for the symbolic execution process itself (instead of input programs) preserving the symbolic semantics (Section V). These optimizations eliminate unnecessary computation recurring in interpreters and are tailored for LLVM's symbolic semantics, leading to smaller and faster generated SE. GENSYM can also perform optimizations to reduce unnecessary forks, *e.g.*, merging cases of identical jump targets when compiling `switch` instructions.

Interpreters could deploy similar optimizations on-the-fly, but at the cost of frequently inspecting program representations or maintaining additional runtime data structures. GENSYM only needs to pay this overhead once at compile time.

Generative Environment Modeling. Real-world programs need to interact with their environment, *e.g.*, invoking syscalls or library functions, for which SE must account. The usual approach is manually building SE *models* of external functions [32], [33]. This is an error-prone and labor-intensive task, which is highly dependent on engine-specific APIs.

GENSYM simplifies model development by using generative programming [15], [17]. Instead of writing models against limited engine APIs in the same language as input programs, we develop models *at the meta level*, *i.e.*, in the same language (and DSL) used to develop our SE compiler. These meta-models describes the symbolic behavior of syscalls in a concise way by using representations of symbolic values, states, and control. Meta-models will be translated to executable code that can be linked with the input program being analyzed. As a prototype, we have developed a symbolic model of the POSIX file system (Section VI).

Evaluation. We first validate the core correctness of GENSYM by examining the tests/paths found by GENSYM on six finite-path programs against KLEE [32], the de facto standard SE engine for LLVM IR written as an interpreter. Experiments show consistent results in all terminating benchmarks.

We evaluate the performance at scale using 14 Coreutils [34] programs involving a standard C library with file system interactions. Our evaluation demonstrates speedups obtained from the synergy of compilation, compile-time optimizations, and parallelism: we observe an average 4.6× speedup of execution time (excluding solver time) compared to KLEE, and speedups up to 9.4× when using 12 threads in parallel.

Finally, we evaluate the effectiveness of compile-time optimizations and the overhead of compilation, showing that our approach can yield net gains in total running time for non-trivial SE tasks.

Contributions.

- We articulate the specification and discuss design considerations of symbolic-execution compilers (Section II).
- We propose to compile SE with continuations and develop GENSYM (Section III) based on partial evaluation. GENSYM is the first SE engine featuring efficient parallel execution (Section IV), optimizations for symbolic semantics (Section V), and executable file system models (Section VI) *all* via code generation.

- We show promising performance speedups on 20 realistic programs by empirically evaluating and comparing GENSYM with LLSC and KLEE (Section VII).

We discuss related work and conclude in Sections VIII and IX.

Availability. GENSYM is publicly available,[1] including the implementation and benchmarks to reproduce the evaluation.

II. Design Space and Considerations

A. Semantics First

Let us first articulate what is a symbolic-execution compiler in the general sense, starting from programming language semantics. For the source language S, we define the *concrete* semantics of S as a collecting trace semantics $[\![\cdot]\!]_S$ [35]. A *trace* $t \in [\![p]\!]_S$ is a sequence of states from a concrete execution instance of the program p.

Symbolic Semantics. Let $[\![\cdot]\!]_S^\#$ be the multi-path symbolic collecting semantics that introduces symbolic values and records path conditions for the source language. Such semantics can be precisely and concisely defined, for example, by an evaluation function [20] or a small-step state-transition relation [36], leading to straightforward interpretation-based implementations. A symbolic semantics usually entails the concrete semantics, since it needs to perform concrete computation as well.

The symbolic semantics extensionally specifies the set of symbolic paths of a program. A *symbolic path* π is a trace that may contain symbolic states and values. A symbolic path can be *concretized* to a set of concrete traces. The symbolic semantics is *sound* if all possible concrete traces t are captured by some symbolic paths from the symbolic semantics:

$$\forall t \in [\![p]\!]_S, \exists \pi \in [\![p]\!]_S^\# \text{ s.t. } t \in \mathit{concretize}(\pi).$$

where $\mathit{concretize}(\pi)$ materializes π to a set of traces, which in practice is performed with the help of SMT solvers [5]. The symbolic semantics is *complete* if all symbolic paths correspond to some concrete execution:

$$\forall \pi \in [\![p]\!]_S^\#, \mathit{concretize}(\pi) \subseteq [\![p]\!]_S.$$

Although the concern of whether $[\![\cdot]\!]_S^\#$ faithfully models $[\![\cdot]\!]_S$ is important, it is orthogonal to compilation.

Symbolic-Execution Compiler. A symbolic-execution compiler (or SE compiler) $\mathcal{C}$ converts a program $p \in S$ to another program $\mathcal{C}(p) \in T$ in language T such that the result of running $\mathcal{C}(p)$ coincides with the symbolic semantics of S. In other words, assuming we have a concrete evaluator $\texttt{eval}_T$ for T, the paths produced by running the generated program, *i.e.* $\texttt{eval}_T(\mathcal{C}(p))$, are equivalent to the paths defined by $[\![p]\!]_S^\#$:

$$\texttt{eval}_T(\mathcal{C}(p)) \simeq [\![p]\!]_S^\#.$$

Under the assumption that the symbolic semantics is sound and complete, we can also characterize the specification of a SE compiler in terms of the concrete semantics of S and T:

$$\forall t \in [\![p]\!]_S, \exists \pi \in \texttt{eval}_T(\mathcal{C}(p)), t \in \mathit{concretize}(\pi) \quad \text{(soundness)}$$
$$\forall \pi \in \texttt{eval}_T(\mathcal{C}(p)), \mathit{concretize}(\pi) \subseteq [\![p]\!]_S \quad \text{(completeness)}$$

[1] https://continuation.passing.style/GenSym

```
x = user_input()
if (x > 42) {
  y = x + 10
} else {
  y = x - 5
}
```

```
pc0 = Set[Constraint]() // init. path constr.
st0 = Map[String, SymValue]() // init. state
st0["x"] = make_symbolic("x")
pc1 = pc0.fork(); st1 = st0.fork() // fork
pc0.add(sym_gt(st0["x"], 42))
st0["y"] = sym_plus(st0["x"], 10)
pc1.add(sym_leq(st1["x"], 42))
st1["y"] = sym_minus(st1["x"], 5)
```

Fig. 1: An example source program (left), and its compiled SE code with deep embedding (right, assuming both paths are feasible). There are two paths $\{x > 42\}$ and $\{x \leq 42\}$. The compiled program collects these paths in `pc0` and `pc1`.

In this sense, "instrumentation-by-compilation" for concolic execution [23], [24] can be considered a special case where we track only one symbolic path in each run, plus the source and the target languages are the same. This paper focuses on the more general case.

B. All About Representations

When designing SE compilers, we must decide how to represent the constructs (*e.g.*, concrete and symbolic values, states, control) of the source symbolic semantics in the target language. This is the central challenge of SE compilers, due to the discrepancy between the symbolic source and non-symbolic target semantics.

Deep vs. Shallow Embedding. We argue that a useful perspective to view this issue is through the lens of DSL embeddings [19], [37], [38], *i.e.*, how we represent one language in the host/target language. Such embeddings can be categorized into "deep" or "shallow". A *deep* embedding uses data structures in T to represent the semantic constructs of S; in contrast, a *shallow* embedding directly uses the semantics of T to encode the desired semantics of S. Deep embedding is useful as it provides explicit representations that can be inspected and transformed, at the cost of another layer of indirection. Shallow embeddings, on the other hand, usually provide better performance due to the avoidance of indirections, but have limited expressiveness.

Values. Primitive concrete values of a source language are straightforwardly represented in shallow embeddings. However, since the target language usually does not have native symbolic values, we have to deeply embed symbolic values and expressions via data structures (*i.e.*, abstract syntax trees) compatible with external SMT solver APIs.

States. Symbolic execution attempts to explore multiple feasible paths *independently* when branch conditions involve symbolic values. Thus, it is convenient to use a deep representation of program states, which are forkable.

Figure 1 shows a simplified example that compiles a branching code snippet into a deep embedding of explicit representations for states and path constraints. Program variables are deeply embedded in the sense that they translate to strings which are the keys of a map data structure assigning symbolic values to each variable (*e.g.*, `st0` and `st1`).

Deep embeddings of states are convenient, but forkable shallow embeddings are also possible and work with the target

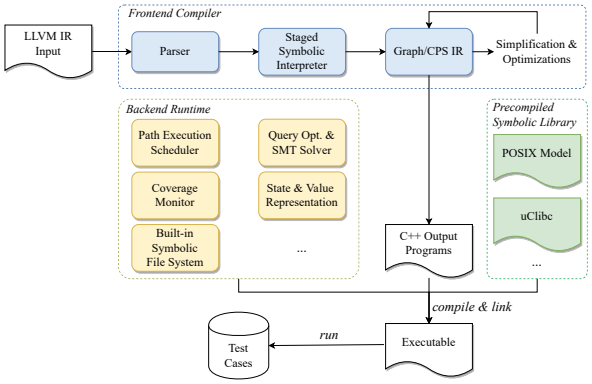

Fig. 2: GENSYM's components and workflow. The POSIX Model and uClibc are adapted from KLEE [32].

language's program variables/states. For instance, EXE [39] uses the `fork` syscall in instrumented programs. However, it makes search or state-merging heuristics hard to implement, since there is little control over forks and no direct manipulation of states.

Control. Implementing searches is simpler with an explicit representation of control in symbolic programs. The simple branching program shown in Figure 1 generates a hard-coded program to explore both branches, assuming they are both feasible. However, considering richer control structures in a low-level IR, every control transfer (branch, jump, and call) could yield multiple states and paths, and we cannot decide path feasibility at compile-time in general. How should we express such nondeterminism in compiled programs?

A simple but "deep" solution is to reify all possible states/-paths into a list and explicitly return the list to the caller (as LLSC [31] does). Nevertheless, this can be very expensive when the number of paths explodes. Instead, GENSYM models nondeterminism "less deeply" by particular uses of *continuations* which abstract over the rest of the execution. To this end, GENSYM transforms programs into continuation-passing style (cf. Section III-C), where continuations are explicitly represented as function values in the target language.

Supporting Heuristics. Search heuristics, although being ad-hoc in nature, are an important ingredient of modern SE. Representations also determine which kinds of search heuristics can be used in the generated code. Scheduling with heuristics often requires pausing or resuming the execution of interesting paths. It pays off to have explicit continuations, since they enable arbitrary control-flow manipulations, and thus provide a general framework for arbitrary heuristics and scheduling policies.

III. GENSYM: DESIGN AND IMPLEMENTATION

A. Workflow

GENSYM is a compiler that translates LLVM IR programs into C++ programs that perform symbolic execution. Figure 2 shows its overall workflow and components:

- The core at the front end is a symbolic interpreter that can be specialized (*i.e.*, partially evaluated) with respect to a parsed LLVM IR program. The specialization constructs the CPS representation of the compiled program.
- The CPS-IR program will be further simplified and optimized, and afterwards a C++ program is generated from it.
- The generated C++ program is then compiled and linked with GENSYM's backend. Optionally, it can be linked with precompiled symbolic libraries (*e.g.*, uClibc [40]).
- Running the final executable generates test cases, which can be further used to measure the real coverage (*e.g.*, by `gcov`).

GENSYM only uses LLVM IR as input and is independent of the LLVM toolchain. While we focus on LLVM IR, the techniques presented here also apply to other languages, including those with structured control statements [20].

B. Compilation by Specialization

Since GENSYM implements compilation by partial evaluation [14], we review the first Futamura projection [21], [22], a fundamental connection between interpreters and compilers.

Partial Evaluation. Let `p(_,_)` be an interpreter with two inputs, `s` the program representation and `d` the program's input. Executing the interpreter yields the value of program `s` with input `d`, *i.e.*, `p(s, d) = v`.

Interpreter *specialization* means dividing the execution of `p` into two stages, according to the availability of inputs or frequency of execution [41], *e.g.*, the program `s` becomes static input at compile time and `d` dynamic input at run time.

A program specializer `spec` can take `p` and `s` as input and produce a *residual* program `target` that has eliminated `p`'s static computation over `s`, *i.e.*, `target = spec(p, s)`. The first Futamura projection states that running the target program with `d` yields the same result as running `p` with both `s` and `d`, *i.e.*, `target(d) = p(s, d)`. We also say `p` is *partially evaluated*, as only the first argument `s` is provided.

Futamura [21], [22] discovered that specializing `p` w.r.t. `s` is equivalent to compilation. This theoretical result is also an effective and practical way to "derive" a compiler from an interpreter (*e.g.*, [42]–[45]).

Our Approach. GENSYM realizes Futamura's first projection without using a dedicated general specializer `spec`. Instead, the symbolic interpreter at its core is lightly annotated with *stages* at the type level [46], [47]. The stage annotations are understood by the Lightweight Modular Staging (LMS) [15] framework, which specializes the staged symbolic interpreter with program `s` and constructs the CPS representation. To get an idea of LMS's stage annotations, consider the following staged `power` function computing b^x in Scala:

```
def power(b: Rep[Int], x: Int): Rep[Int] =
  if (x == 0) 1 else b * power(b, x - 1)
```

The stage annotation `Rep[T]` is used as the type of `b` and the result. It indicates that the current-stage value of `Rep[T]` is a *rep*resentation of a next-stage `T` value. Moreover, the function body requires no changes whatsoever, making this approach to staging lightweight. LMS can specialize `power` given a statically known value for `x`, *e.g.*, when `x = 3` we have the following generated program:

```
def power3(b: Int): Int = b * b * b
```

This design allows us to express the symbolic semantics of LLVM IR in the form of a staged interpreter [20], a high-level and easy-to-understand artifact, and then automatically obtain a compiler that produces efficient and performant code. The program IR is statically known, thus not marked with `Rep` type in the interpreter, whereas the program states (*i.e.*, memory, PC, etc.) are represented for future manipulation.

C. Compiling Symbolic Semantics of LLVM IR

The essence of GENSYM compilation is to transform source programs to CPS with symbolic semantics by specializing the staged symbolic interpreter that operates on an explicit representation of program state (stack/heap) and control (continuations) [48].

Compiling Values. LLVM's concrete values are integers and floating-point numbers, which compile to corresponding C++ values. We also introduce symbolic values depending on supported theories of the solver, and model symbolic integers using fixed-size bitvectors. Nevertheless, richer solver-supported datatypes can be readily added. Primitive operations (*e.g.*, arithmetic) translate to back-end operations that work for both concrete and symbolic values.

Compiling States. During compilation, GENSYM makes program states explicit so that every effectful LLVM instruction operates on a state. An important aspect of states is they can be forked, and thus our state representation is *deep*, *i.e.*, by using data structures in the target language.

For example, an LLVM assignment instruction to a local variable is directly translated to a C++ method call on the current state `s`:

```
z = add i32 42 y          →    s.assign("z", IntV(42) + s.lookup("y"))
```

Local variables mentioned in this instruction are translated to a lookup. In Section V, we discuss how to eliminate redundant assignments and lookups. Memory operations such as `store` and `load` compile analogously:

```
store i32 z, i32* ptr     →    s.store(s.lookup("ptr"), s.lookup("z"))
a = load i32, i32* ptr    →    s.assign("a", s.load(s.lookup("ptr")))
```

GENSYM's runtime implements several state operations to manipulate the stack, heap, and local frames. The interfaces of these operations are exposed to the symbolic interpreter for representing and generating code.

Compiling Control. Now we discuss why CPS is the key in compiling SE. Consider the following LLVM `call` instruction.

```
ret = call i32 f(...)
n   = ...  ; some computation using ret
```

During symbolic execution, function `f` could explore an arbitrary number of paths internally. Thus, each of these paths yields its own return value bound to `ret`, and each value is used by an independent instantiation of the rest of the computation after the call to `f`, *i.e.*, the program's *continuation*.

We make the continuation explicit via the CPS translation that reifies it into a first-class function, which takes an argument representing the return value of `f`. We transform the function call site to take the continuation as an argument, *i.e.*, `f(..., k)`. The function body of `f` will be transformed as well:

```
1 /* C source */
2 int power(int x, int n) { if (n == 0) return 1; return x * power(x, n-1); }
3 /* LLVM IR */
4 define i32 @power(i32 x, i32 n) {
5   b0: cmp = icmp eq i32 n, 0
6       br i1 cmp, label b2, label b1
7   b1: subv = sub i32 n, 1
8       retv = call i32 @power(i32 x, i32 subv)
9       mulv = mul i32 x, retv
10      br label b2
11  b2: r = phi i32 [mulv, b1], [1, b0]
12      ret i32 r
13 }
14 /* GenSym's generated code; S/V are runtime state/value type */
15 using Cont = function<void(S, V)>;  // the continuation type
16 void power_b0(S s, List[V] args, Cont k) { // compiled block 'b0'
17   s.assign_seq(List("x", "n"), args);
18   V cmp = args[1] == IntV(0L, 32);
19   s.set_cur_block("b0"); // record the current block (for phi node)
20   if (cmp.is_conc()) {   // concrete branch condition
21     if (cmp.value() == 1) power_b1(s, k); // jump to block 'b1'
22     else power_b2(s, k);                  // jump to block 'b2'
23   } else { // symbolic branch condition
24     // yield the control to scheduler and add two new tasks to schedule
25     schedule(task(s, cmp, power_b2, k), task(s.fork(), !cmp, power_b1, k));
26   }
27 }
28 void power_b1(S s0, Cont k) { // compiled block 'b1'
29   V subv = s0.lookup("n") - IntV(1, 32);
30   List[V] args = List(s0.lookup("x"), subv);
31   // recursively call 'power' with a new continuation
32   power_b0(s0, args, [=](S s1, V retv) {
33     s1.assign("mulv", retv * s1.lookup("x"));
34     s1.set_cur_block("b1");
35     power_b2(s1, k);
36   });
37 }
38 void power_b2(S s, Cont k) { // compiled block 'b2'
39   V r = s.last_block() == "b1" ? s.lookup("mulv") : IntV(1L, 32);
40   k(s, r);
41 }
```

Fig. 3: The C, LLVM IR, and GENSYM's (simplified) generated code for the recursive `power` function.

instead of directly "returning" some value from `f`, we invoke the continuation `k(v)` where `v` is that returned value. Therefore, if there are multiple returning paths, they each invoke the same continuation with their own results.

In addition, each continuation in GENSYM takes a state as an argument, so that each execution path has its own fork of the (deeply embedded) symbolic program state. This ensures that invoking the same continuation multiple times has no unintended side effects.

A Concrete Example. Figure 3 shows the `power` function as (1) C source code, (2) its LLVM IR form (the input to GENSYM), and (3) the symbolic code produced by GENSYM. There are 3 blocks in the `power` function (`b0`, `b1`, and `b2`) and we compile each of them to a C++ function in the generated code. The compiled function for the entry block `b0` takes a state, a list of argument values, and a continuation. Non-entry block functions only take a state and a continuation. Continuations are represented by the type `function<void(S, V)>` in C++ (line 15) taking a state and a value.

Basic LLVM control instructions compile straightforwardly:

- Direct jumps between blocks become function calls with the current state and continuation.
- Branches are compiled to first perform a concreteness check (line 20) of the condition. If it is concrete (line 21-22), we deterministically execute one branch depending

on the concrete value; otherwise, we yield control to the scheduler (line 25) and let it decide which path to explore. We pack the continuation `k` into tasks, indicating the two forked paths sharing the same remaining computation. A naive scheduler can be realized by executing the two tasks in order (*i.e.*, DFS search). In Section IV, we discuss how scheduling works with parallelism.

To explain function-call compilation, consider the recursive call of `power` at line 8. In its compiled code (line 32-36), the continuation is an anonymous function prepared by the caller and passed into the callee (line 32). The continuation's argument `retv` represents the returned value from the call. Thus, in the body of the continuation (line 33-35), we use `retv` to continue the computation after the callee returns. At the end of the continuation, we transfer control to block `b2` with the outer-level continuation `k`, chaining the rest of the computation. Since continuations are callable first-class objects, the callee (*i.e.*, `block_b0`) can directly call it multiple times (*i.e.*, return multiple values) or store it into the scheduler for later invocation, as shown in this example.

The LLVM `ret` instruction is compiled to invoking the current continuation with the returned value (*e.g.*, in block `b2`, line 40). The compiled code also maintains some meta information, *e.g.*, the currently executing block (`set_cur_block`, which is used to resolve `phi` instructions) and coverage information (not shown in Figure 3 for brevity).

D. Implementation

GENSYM supports most of the common LLVM instructions and symbolic operations for integers. GENSYM's memory model precisely keeps track of both concrete and symbolic values with bit-precision. It also supports calling indirect functions, reading symbolic pointers, and tracking bounds of allocations, which are essential to execute general C programs.

The front-end GENSYM compiler is written in Scala, consisting of ~3K LOC. The core staged symbolic interpreter is still quite concise, implemented within ~400 LOC using LMS [15]. The core runtime implementation is written in C++ (~4K LOC), including the data structures of values and states, path scheduler, coverage monitoring, and solver support. The back-end implementation makes use of persistent data structures [49] to represent memory, allowing low-overhead structural sharing and non-interference between multiple concurrent/parallel paths.

IV. PARALLELISM

In practice, the number of paths in SE easily grows exponentially, even on small- to medium-sized programs. Next to using clever heuristics to select paths, using more hardware resources is an effective complementary solution to the path explosion problem. Prior studies have explored *interpretation-based* approaches with distributed clusters [33], client-server frameworks [50], or multi-threads [51], which have complex architectures and incur considerable communication and interpretation overhead. GENSYM explores this direction by utilizing its unique CPS-based compilation schema and multi-core CPUs on a single machine:

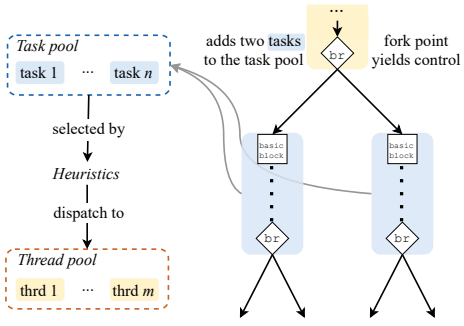

Fig. 4: GENSYM's backend parallel execution architecture and the execution flow as a fork tree in the compiled code.

- The compiled SE tasks are executed cohesively in a single address space with shared memory, incurring lower communication overhead and no interpretation overhead.
- Thanks to CPS, the compiled program can execute multiple paths in parallel and integrate custom search heuristics, with the cooperation of a scheduler.

Cooperative Concurrency/Parallelism. Figure 4 visualizes how GENSYM explores multiple paths in parallel. The back-end *scheduler* maintains (1) a task pool containing unexplored paths/states, and (2) a thread pool containing a fixed number of workers that can be specified at run-time.

At a fork point, the current thread adds new unexplored paths to the scheduler before yielding control to it. The added tasks capture the current continuation representing the rest of the execution (recall line 25, Figure 3). In fact, by generating CPS code we can yield control at any point and suspend the rest of the computation in the form of a continuation.

When a thread is running, it executes a "*slice*" (light blue in Figure 4) of a path in the whole SE fork tree. This "slice" is a branch-free atomic task consisting of sequential blocks. When the task finishes at a fork point, its running thread attempts to execute the next available path, selected by some heuristic.

Integrating Heuristics. As search heuristics must interact with the scheduling of paths, our CPS-based approach offers a simple and effective way to integrate search heuristics with compilation. We have implemented strategies including random path and random state search. Orthogonal strategies to partition paths for parallel SE (*e.g.*, [33], [50]) can also be integrated. For instance, we could have multiple schedulers, each exploring certain regions in the execution tree and dispatching among its assigned threads.

Without explicit control representations as in CPS, one must use heavyweight mechanisms to implement parallelism in compiled programs. It also prevents switching to a scheduler while executing some path, *e.g.*, LLSC [31] uses C++'s `async` as a parallelism primitive, which cannot guarantee the resource utilization. Thus, it is overall less scalable compared to GENSYM. In Section VII-C, we evaluate the performance of GENSYM's parallel execution and show significant speedups.

V. Optimizations

Under the "two-stage" execution model (Section III-B), we also distinguish between compile-time and run-time optimizations in GenSym. This section highlights selected compile-time optimizations, some of which are performed *online* in interpreters exhibiting recurring overhead. We will not discuss the run-time optimizations (*e.g.*, hash consing, query caching, constraint independence, etc.), which are standard and also found in other engines.

Lookup Elimination. In an SSA IR (*e.g.*, LLVM), local variables (*i.e.*, virtual registers) are assigned only once. For example, at line 5 in Figure 3, the result of a comparison is assigned to register `cmp`. Interpretation-based engines need to maintain a map from variables to values and constantly look up variables when executing instructions referring to them.

However, most of these lookups are redundant since we can locally resolve their values at compile-time. This is safe since each register is assigned only once. A lookup operation for identifier x can be reduced by traversing over the GenSym-IR to find the value assigned to x:

```
s.assign(x, v1)                 s.assign(x, v1)
...                      →      ...
s.lookup(x)                     v1
```

From the perspective of embeddings (Section II-B), this optimization shifts a deep embedding of *variable bindings* to a shallow embedding.

Assign Elimination. Exhaustive elimination of lookups immediately suggests that dead assignments of registers can be eliminated as well. We can inspect the GenSym-IR and eliminate those unneeded assignments per function scope:

```
s.assign(x, v1)                 ~~s.assign(x, v1)~~
... /* no lookup of x */   →    ...
```

Source program optimization `mem2reg` [52] performs a similar job, reducing the memory mapping but not the register binding map. For instance, the LLVM program in Figure 3 is already preprocessed by `mem2reg` and still has the assignment of `cmp`, which is eliminated by GenSym after compilation.

Compile-time State Merging. A simple way to compile LLVM's `switch` is translating it to a series of binary branches. However, this often amplifies path explosion, especially for cases that share the same target block. Therefore, instead of compiling `switch` instructions naively (left), GenSym can generate a single guard by a disjunction of the conditions:

```
if (c1)  tgt(...)
if (...) tgt(...)          →    if (c1 || ... || cn) tgt(...)
if (cn)  tgt(...)
```

This is a simple form of static state merging [53] to reduce forking overhead. Symbolic interpreters can indeed perform similar optimizations at run-time (*e.g.*, as in KLEE). However, they have to perform this transformation recurringly, even when re-encountering the same `switch` instruction.

Algebraic Simplification. Symbolic expression simplification is another compile-time optimizations in GenSym, which may affect the solver performance. For example, we observe a pattern in source programs where comparisons are made atop bit-vector (BV) extensions (which can often be expensive for solvers), even on source operands of the same width. GenSym attempts to decide if the widths of BVs before extension are identical. In that case, GenSym would only generate eq expressions:

$$\mathsf{eq}(\mathsf{ext}(v_1, bw), \mathsf{ext}(v_2, bw)) \quad \rightarrow \quad \mathsf{eq}(v_1, v_2)$$

GenSym deploys many similar optimizations exploiting the equational theory behind these algebraic operations.

Discussion. The demonstrated compile-time optimizations help generate faster and smaller code. But they are not exhaustive: Optimizing SE by code generation is still largely underexplored. We expect more analyses/transformations that are informed by the static program structure can be naturally integrated into our two-stage execution model.

VI. Environment Modeling

GenSym can either use KLEE's file system model as part of input programs, or its own built-in generative symbolic file system (Figure 2).

Reusing KLEE's Model. KLEE has a POSIX file system model written in C against its APIs. This model should be linked with the analyzed program before running the KLEE engine. During execution, KLEE intercepts those API calls and performs corresponding symbolic operations. GenSym can benefit from reusing KLEE's file system by exposing similar APIs to the model and compile it as a normal program.

However, modeling in low-level languages can be labor-intensive, error-prone, and suboptimal performance-wise. Since the model is written in C and conceptually a "user" program, it needs further translation to be executed, incurring additional overhead. The model can only communicate with APIs exposed by the engine. This stratification limits the model's capability to manipulate states and control.

Generative Meta-level Modeling. We therefore explore a fused approach to integrate models with the engine at the meta level, using higher-level programming abstractions. Models are written in the same language (Scala) we use to develop our staged symbolic interpreter, which has direct access to symbolic values, states, and control representations. Using the same code generation technique (Section III), meta-models are converted to C++ code which is linked with GenSym's backend and can be used by symbolic programs. This approach is not only easier to develop, but also provides better performance, due to the elimination of indirections. In this way, we have implemented prototype models for 13 file system calls, including directory support.

VII. Empirical Evaluation

In this section, we aim to answer the following research questions through empirical evaluation.

RQ1: Does GenSym generate correct code to perform SE on LLVM IR?

RQ2: How is the single-thread execution performance of GenSym compared with prior works?

RQ3: How is the parallel execution performance of GenSym?

RQ4: Is it practical to use GenSym after considering its compilation overhead?

RQ5: What is the impact of our compile-time optimizations?

TABLE I: Single-thread performance (in sec.) on algorithm benchmarks. "C/LLVM LOC" describe sizes of input programs excluding comments/blanks. T_{Solver} includes query caching/optimization time. $T_{\text{Solver}} + T_{\text{Exec}}$ is end-to-end wall time T_{Whole}.

	Description				LLSC			KLEE			GENSYM			T_{Exec} Speedup		T_{Whole} Speedup	
	C LOC	LLVM LOC	#Sym Args	#Paths	T_{Solver}	T_{Exec}	T_{Whole}	T_{Solver}	T_{Exec}	T_{Whole}	T_{Solver}	T_{Exec}	T_{Whole}	vs. LLSC	vs. KLEE	vs. LLSC	vs. KLEE
bubble sort	38	144	6	720	56.85	0.321	57.17	23.19	0.259	23.45	10.75	0.055	10.81	**5.83x**	**4.70x**	**5.29x**	**2.16x**
KMP matcher	65	254	10	4181	5.99	0.568	6.56	0.443	0.320	0.763	0.094	0.038	0.128	**14.95x**	**8.42x**	**51.25x**	**5.96x**
knapsack	40	171	4	1666	156.58	0.459	157.04	204.08	0.807	204.88	263.35	0.107	263.46	**4.29x**	**7.54x**	**0.60x**	**0.78x**
merge sort	71	337	7	5040	70.54	1.06	71.60	70.04	1.32	71.36	21.44	0.188	21.63	**5.64x**	**7.02x**	**3.31x**	**3.30x**
nqueen	56	308	25	1363	-	-	-	0.652	4.73	5.38	1.08	0.793	1.87	**-**	**5.96x**	**-**	**2.88x**
quick sort	44	142	7	5040	112.47	1.40	113.87	149.22	2.35	151.57	39.78	0.293	40.07	**4.78x**	**8.02x**	**2.84x**	**3.78x**

To answer RQ1, we compare the outcomes of GENSYM runs against KLEE [32] and LLSC [31]. To answer RQ{2,3}, we conduct a performance evaluation and compare GENSYM with KLEE and LLSC under various settings. To answer RQ4, we report the compilation time and reveal its viability. To answer RQ5, we evaluate the overhead and consequence (size and running time of generated code) of these optimizations. We discuss threats to validity in Section VII-F.

Environment and Setup. All experiments are conducted on a machine with 4 Intel Xeon 8168 CPUs and 3TB memory, running Ubuntu 20.04 with kernel 5.4.0. The machine is a NUMA (non-uniform memory access) machine with 96 physical cores in total (8 nodes). `numactl` is used to set CPU and memory affinities of processes. We use `clang` 11 to generate source LLVM IR from C programs and `g++` 9.4.0 `-O3` to compile the code generated by GENSYM. KLEE (ver. 2.3) and GENSYM use the Z3 solver [54] (ver. 4.8.12). LLSC (brach `fse21demo`) only supports STP [55] (ver. `ac1b92b`).

Benchmarks. The first set of benchmarks (Table I) are algorithm implementations with fixed input sizes. They are small but realistic programs consisting of finite paths and covering the core semantics of LLVM IR. We use this benchmark set to validate the correctness (RQ1) of GENSYM. We also report the running times on these benchmarks for RQ2.

We further use a subset of the GNU Coreutils v8.32 programs for a larger-scale performance evaluation:

```
base32 base64 cat comm fold echo dirname
expand paste  cut join link true pathchk
```

These programs are nontrivial and representative examples of real-world programs that exercise important features such as interaction with the C library, with the shell environment (*e.g.*, handling command line arguments), and with the file system (*e.g.*, reading/writing to stdin/stdout and files). Therefore they are linked with KLEE's POSIX FS system model [56] and uClibc library [40]. We generate LLVM IR (`.ll`) from the KLEE POSIX and uClibc source code, so that they can be used as inputs to GENSYM. The average LLVM IR size of linked Coreutils program is 28334LOC.

C-to-LLVM generation is performed by `clang` with `-O0`, along with preprocessing steps used in KLEE (*e.g.*, clean up intrinsics and inlined assembly, etc.) to make sure both engines take the same input programs. To ensure accurate path number counting, both engines treat `switch` instructions as non-mergeable branches.

A. RQ1: Validating Correctness

This experiment is concerned with the correctness of the compiled SE, *i.e.*, whether GENSYM generates code that does not explore any spurious path and does not miss any true path. Since formally verifying the correctness of SE compilers would be another challenging issue, this is conducted by empirically examining the coverage and the generated tests. All SE engines are expected to achieve 100% path coverage. The differences in SMT constraint encoding and search heuristics of the engines should not bring in different results.

Results. As expected, all engines report 100% path coverage for the benchmarks listed in Table I (excluding `nqueen` where LLSC fails on an unsupported instruction). This validates that GENSYM generates correct code regarding core LLVM instructions.

B. RQ2: Single-Thread Execution Performance

We compare the single-thread performance of GENSYM with LLSC and KLEE. The reported running times are divided into "solver time" and "execution time". The former accounts for caching, optimizing, and solving queries, and the latter for scheduling and executing paths, summing up to the total running time (wall time). Our focus is evaluating *execution time* (T_{Exec}).

Comparison with LLSC. Since LLSC lacks support of file systems, we can only compare GENSYM with LLSC on the algorithmic benchmark set. Table I shows the result.

First, we clarify the differences in solver time characteristics. With similar state and value encodings, LLSC uses STP and deploys no solver optimization, while GENSYM uses Z3 and simplifies queries before solving them. Nevertheless, solver time improvement is not a contribution of GENSYM.

For execution time, we observe GENSYM to be 7× faster than LLSC on average. While both tools are compilers, their difference stems from how they represent nondeterminism in path exploration. GENSYM uses CPS and invokes continuations multiple times, whereas LLSC reifies and manages a list data structure for alternatives, which is more expensive.

Comparsion with KLEE. In addition to the algorithm benchmark set, we use Coreutils (with POSIX/uClibc) benchmarks to compare GENSYM and KLEE. Both engines use a random path search strategy and output tests only for new states discovered. KLEE applies its default query optimizations pipeline, while GENSYM only uses hash consing, query caching, and constraint independence resolving.

On algorithm benchmarks (Table I), we observe an average of 6.9× speedup in GENSYM's execution time. The greatest

TABLE II: Single-thread performance (in sec.) on Coreutil programs. sym-stdin/sym-files/sym-arg indicate the number of symbolic inputs in stdin/files/CLI arguments ($n \times m$ means n files with m symbolic values; $n + m$ means two symbolic arguments of size n and m). Both engines use symbolic stdout. T_{Query} is the actual constraint solving time (included in T_{Solver}).

Benchmark Configuration				KLEE						GENSYM							
	sym-stdin	sym-files	sym-arg	Path Cov	Line Cov	T_{Query}	T_{Solver}	T_{Exec}	T_{Whole}	Path Cov	Line Cov	T_{Query}	T_{Solver}	T_{Exec}	T_{Whole}	T_{Exec} Speedup	T_{Whole} Speedup
base32	2	2×2	2	10521	73.33%	3.21	5.77	47.38	53.15	10621	73.33%	28.72	33.97	12.54	46.51	**3.78x**	**1.14x**
base64	2	2×2	2	10524	73.33%	3.26	5.83	47.50	53.33	10624	73.33%	28.89	34.12	12.49	46.61	**3.80x**	**1.14x**
cat	2	-	2	29151	80.58%	5.07	9.88	126.34	136.22	28539	80.91%	25.80	34.53	28.49	63.02	**4.43x**	**2.16x**
comm	2	2×2	2+1	23846	70.11%	6.44	11.59	107.29	118.88	23846	72.3%	32.37	42.31	28.21	70.52	**3.80x**	**1.69x**
cut	2	2×2	2+2	28558	72.09%	6.05	11.20	129.58	140.79	28481	65.86%	16.63	21.54	33.48	55.01	**3.87x**	**2.56x**
dirname	2	-	6+10	287386	100.0%	1.29	18.28	341.51	359.79	287386	100.0%	2.61	10.98	93.84	104.83	**3.64x**	**3.43x**
echo	-	-	2+7	216136	84.17%	2.55	9.02	239.26	248.27	216136	84.17%	2.58	4.46	63.05	67.52	**3.79x**	**3.68x**
expand	2	2×2	2	10870	72.37%	4.18	6.85	49.11	55.96	10870	71.05%	37.65	42.98	13.03	56.01	**3.77x**	**1.0x**
true	-	-	10	16	100.0%	0.22	0.23	0.06	0.29	16	100.0%	0.02	0.04	0.03	0.07	**2.00x**	**4.14x**
fold	2	2×2	2	11015	74.36%	4.12	6.86	49.46	56.32	11015	74.36%	37.37	42.79	13.12	55.91	**3.77x**	**1.01x**
join	2	2×2	2+1	25054	71.75%	11.14	16.77	113.05	129.82	25046	70.93%	94.56	105.18	30.15	135.33	**3.75x**	**0.96x**
link	2	2×2	2+1+1	11233	60.0%	2.85	6.20	74.72	80.93	11233	60.0%	6.89	17.90	21.11	39.01	**3.54x**	**2.07x**
paste	2	2×2	2+1	24760	76.08%	7.10	12.28	110.96	123.23	22622	76.08%	36.27	45.91	26.41	72.32	**4.20x**	**1.70x**
pathchk	2	2×2	2+2	10923	63.31%	6.69	11.95	50.57	62.52	10923	63.31%	35.50	46.20	15.32	61.51	**3.30x**	**1.02x**

	sym-stdin	sym-files	sym-arg	Path Cov	Line Cov	T_{Query}	T_{Solver}	T_{Exec}	T_{Whole}	Path Cov	Line Cov	T_{Query}	T_{Solver}	T_{Exec}	T_{Whole}	Path Throughput Ratio
base32	4	2×2	4	565835	90.83%	37.28	166.50	3434.22	3600.72	1104408	90.83%	848.75	1729.35	1377.32	3106.67	**4.87x**
base64	4	2×2	4	569296	90.83%	36.15	159.98	3440.71	3600.69	1104674	90.83%	1051.77	1933.35	1351.34	3284.69	**4.94x**
cat	3	-	3	715174	85.12%	9.93	127.60	3482.59	3610.19	2549614	85.12%	83.06	1002.99	2598.26	3601.25	**4.78x**
comm	3	2×2	3+1	565438	90.8%	12.37	119.38	3480.96	3600.34	1913738	91.38%	88.12	1154.43	2446.83	3601.26	**4.81x**
cut	3	2×2	3+3	551059	88.84%	31.47	153.85	3447.76	3601.61	2232944	86.05%	241.00	840.13	2761.12	3601.26	**5.06x**
dirname	6	-	9+15	4063657	100.0%	1.87	260.52	3388.31	3648.83	12838901	100.0%	6.87	341.76	3259.63	3601.39	**3.28x**
echo	-	-	4+8	2712142	84.17%	3.36	120.26	3496.94	3617.20	10952611	84.17%	3.41	143.68	3458.14	3601.82	**4.08x**
expand	3	2×2	3	92478	92.11%	3107.99	3134.37	466.45	3600.81	1062845	90.79%	60.10	718.39	1287.12	2005.51	**4.17x**
fold	3	2×2	3	567926	97.44%	20.20	139.38	3460.90	3600.28	1099906	97.44%	468.87	1130.24	1417.35	2547.59	**4.73x**
join	3	2×2	3+3	582806	80.41%	60.67	194.64	3409.56	3604.20	2246226	76.7%	69.75	808.07	2793.69	3601.76	**4.70x**
link	3	2×2	3+3+3	552717	80.0%	9.06	156.73	3451.60	3608.33	1241043	80.0%	25.94	1277.89	2323.53	3601.42	**3.34x**
paste	3	2×2	3+3	517354	92.82%	563.83	776.72	2825.86	3602.58	2020402	92.82%	376.65	1165.32	2436.07	3601.39	**4.53x**
pathchk	3	2×2	3+3	537795	71.22%	402.34	724.59	2875.70	3600.30	1132308	71.22%	172.57	1780.38	1821.11	3601.49	**3.32x**

speedup is 8.4× on `KMP matcher`. Notably, KLEE spends a dominant execution time of 4.73s on `nqueen`, whereas GENSYM uses only 0.79s.

Then, we use two configurations to test Coreutils programs. The short-running configuration (upper, Table II) has fewer symbolic inputs and both engines perform a similar job and terminate in a few minutes, thus we can faithfully compare execution time speedup. The long-running configurations (lower, Table II) has more symbolic inputs and we set a 1-hour timeout, thus we compare the throughput assuming paths are homogeneous. Path throughput is computed by comparing the number of paths explored by GENSYM and KLEE per second (excluding the solver time). To validate effectiveness, we also report path and line coverages (from `klee-replay/gcov`).

In Table II (upper), we first notice that GENSYM has similar path coverage (hence line coverage) as KLEE on most cases. The slight mismatches are caused by different concretization strategies. Overall, we observe an average of 3.7× speedup in execution time. Moreover, speedups are consistent, indicating a constant advantage over KLEE. In Table II (lower), KLEE times out on all benchmarks, whereas GENSYM finishes `base32`, `base64`, `fold`, and `expand` within timeout. On all cases we observe that GENSYM explores more paths than KLEE, although it spends less time in execution and more time in the solver chain. With more paths explored, GENSYM however does not achieve higher line coverage on these programs using a random path selection heuristic. We expect a smarter heuristic could use GENSYM's power more effectively. To conclude, GENSYM explores 4.7× as many paths in a unit time as KLEE does in the long-running experiment.

Query/Solver Time. Table II reports a separate *query time* (T_{Query}), which is the time spent in actual constraint solving (*i.e.*, in Z3). Query time is included in solver time. Although query/solver time is not within the scope of our contribution, we explain the difference in observed query/solver times.

First, GENSYM uses bit-vectors to encode constraints and resolves memory reads/writes within the engine, whereas KLEE uses symbolic arrays. GENSYM has not aggressively simplified the query before sending to the solver, therefore GENSYM tends to spend more time in T_{Query}. We however notice that different solvers may behave quite differently, *e.g.*, GENSYM's T_{Query} is much less when using STP.

Second, KLEE's solver-chain time ($T_{\text{Solver}} - T_{\text{Query}}$) is shorter than GENSYM's, especially on long-running benchmarks. KLEE deploys well-tuned solver optimizations (*e.g.*, query caching, equality rewriting, constraint independence), which exhibit better efficiency under higher pressure. In contrast, GENSYM's implementation of solver optimizations is still effective (since query time is reduced vs. optimizations disabled), but less efficient in larger Coreutils benchmarks.

The solver-chain implementation is an orthogonal issue to GENSYM compilation schema, given that the paths explored by both tools are similar. We expect that the solver-chain performance can be improved significantly by adopting optimizing implementations.

Results. We have consistently observed speedups in execution time (7× vs. LLSC, 4.6× vs. KLEE) on all 20 benchmarks. Even with the solver time considered, GENSYM is on average 2.4× faster than KLEE on these benchmarks. There are 2 (out of 20) programs where GENSYM is slightly slower than KLEE end-to-end.

C. RQ3: Parallel Execution Performance

We use the Coreutils benchmarks and short-running configurations as in RQ2 to evaluate GENSYM's parallel execution

TABLE III: Speedups of parallel execution (1-thread as 1.0x baseline).

	With Solver Opt			Without Solver Opt		
	4 th	**8 th**	**12 th**	**4 th**	**8 th**	**12 th**
base32	2.12x	2.69x	2.72x	3.64x	6.95x	9.51x
base64	2.15x	2.70x	2.58x	3.97x	7.03x	9.82x
cat	1.89x	2.60x	2.89x	3.80x	7.58x	10.47x
comm	1.88x	2.50x	2.73x	3.68x	7.02x	9.74x
cut	2.09x	3.09x	3.76x	3.74x	7.16x	10.44x
dirname	2.53x	3.87x	4.54x	3.33x	5.89x	8.47x
echo	2.38x	3.50x	4.13x	3.59x	6.81x	9.82x
expand	2.13x	2.83x	2.83x	3.39x	6.11x	7.97x
fold	2.23x	2.92x	3.05x	3.60x	6.43x	8.74x
join	1.84x	2.62x	3.09x	3.48x	6.29x	8.79x
link	2.14x	2.80x	3.04x	3.66x	6.81x	9.49x
paste	1.95x	2.64x	2.86x	3.64x	6.55x	8.95x
pathchk	2.08x	2.84x	3.10x	3.54x	6.50x	8.85x
average	**2.08x**	**2.83x**	**3.10x**	**3.63x**	**6.74x**	**9.36x**

performance and compare it with single-threaded execution. Under parallel execution model, each worker uses an independent SMT solver, and caching or independence constraint resolving result are not shared among solvers. Moreover, a global dispatcher (*e.g.*, by selecting a random path) is synchronized and guarded by locks. We run 4/8/12 threads within a NUMA node, which consists of 12 physical cores.

Results. We show the parallel execution speedups in Table III. Overall, we observe an average 2.08×/2.83×/3.10× speedup when using 4/8/12 threads vs. the single-thread baseline. Considering the increasing amount of resources, short-running benchmarks do not exhibit ideal speedup ratios. Since the single-thread execution is already relatively short (less than 1 minute for most of the benchmarks), using more threads would not deliver higher performance.

Another bottleneck preventing higher scalability again lies in the query optimization, since each worker uses a separate solver instance with isolated caching/optimization facilities. We expect that the efficiency can be further improved if the solver chain or query optimizations can be shared.

We further experiment with disabling query-chain optimizations (right half of Table III), which more faithfully evaluates the efficiency of continuation-based parallel execution. The average speedups 3.63×/6.74×/9.36× exhibit higher speedup/resource ratio and confirms the efficiency of the continuation-based parallel execution.

D. RQ4: Compilation Cost

Being a prototype for generating optimized code, GENSYM itself has not yet been optimized to reduce compilation time. Highly optimized code is achieved often at the cost of increased compilation time. Still, we show that it is practical to use GENSYM with the cost considered. As a compiler, GENSYM can benefit from separate compilation. To compile large programs such as Coreutils, we first prepare the POSIX/uClibc library as an LLVM module (98081 LOC). GENSYM compiles it to C++ and further to a binary library, which can be reused by any application relying on it. Symbols are resolved during code generation, leaving no overhead at runtime.

Results. GENSYM takes 78s to compile the POSIX/uClibc library and an additional 4min10s to generate the binary from C++ code using 96 physical cores in parallel. Although relatively large, this is an entirely one-time effort. With the libraries prepared, an application such as `echo` (LLVM IR size 6922 LOC, without library code) takes 1.54s to generate C++ code and 44.36s for the final executable using parallel compilation, as reported in Table IV. In contrast, compiling `echo` with libraries linked (LLVM IR size 29448 LOC) takes 37s to generate C++ and 1m2s to build the executable. The cost is worth paying regarding the runtime speedup shown in RQ2, especially considering repeated runs on larger problem sizes. This is in accordance with other applications presented in Table IV. The compilation overhead can be further alleviated by existing techniques, such as pre-building the header-only runtime, which is mostly an engineering effort and orthogonal to our contribution.

TABLE IV: Evaluation of compilation time (RQ5) and compile-time optimizations (RQ6). All timings are shown in seconds. Code size and $T_{\textbf{exec}}$ compares optimized and unoptimized versions.

	$T^{\text{w/o opt}}_{\text{IR}\to\text{C++}}$	$T^{\text{w/o opt}}_{\text{C++}\to\text{bin}}$	$T^{\text{w/ opt}}_{\text{IR}\to\text{C++}}$	$T^{\text{w/ opt}}_{\text{C++}\to\text{bin}}$	Code Size	$T_{\textbf{exec}}$
base32	1.51	51.99	2.12 (+0.61)	49.36 (-2.63)	-15.90%	-12.05%
base64	1.47	51.25	2.05 (+0.59)	48.28 (-2.97)	-15.68%	-9.97%
cat	1.65	48.73	1.99 (+0.34)	46.68 (-2.05)	-15.27%	-17.49%
comm	1.50	50.56	2.27 (+0.77)	49.14 (-1.41)	-16.05%	-15.95%
cut	1.58	51.80	2.25 (+0.66)	48.43 (-3.37)	-16.55%	-21.24%
dirname	0.99	46.06	1.34 (+0.35)	44.32 (-1.74)	-15.45%	-32.41%
echo	1.15	46.54	1.54 (+0.39)	44.36 (-2.17)	-15.00%	-36.84%
expand	1.23	50.72	1.73 (+0.50)	48.17 (-2.55)	-15.08%	-10.58%
fold	1.36	49.25	2.24 (+0.88)	44.58 (-4.67)	-14.90%	-9.92%
join	2.88	56.06	3.61 (+0.73)	53.32 (-2.74)	-15.92%	-10.37%
link	0.93	47.22	1.28 (+0.35)	45.29 (-1.93)	-15.50%	-21.58%
paste	1.27	48.53	1.73 (+0.47)	46.24 (-2.29)	-15.40%	-14.62%
pathchk	1.21	48.84	1.67 (+0.46)	46.29 (-2.55)	-15.52%	-36.37%
true	0.94	45.58	1.60 (+0.66)	43.22 (-2.36)	-15.42%	-0.15%

E. RQ5: Overhead & Effectiveness of Optimizations

So far, all experiments are conducted with compile-time optimizations enabled. In this experiment, we evaluate the overhead and performance with and without our compile-time optimizations. Table IV shows the impact of these optimizations on the Scala-end time in emitting the C++ code, the size of the generated C++ code (LOC), the time in compiling the C++ code (96 cores in parallel), and the execution time of the built code. Atop the optimizations discussed in Section V, we exclude the compile-time state merging for faithful path counting, and additionally incorporate ordinary optimizations including constant folding, dead code elimination, and function inlining. We use the short-running configurations of Coreutils programs (upper Table II) to measure the execution time impact. The unoptimized applications are linked with an unoptimized version of the prebuilt library, which takes about 59s in code emission and 4min26s in parallel C++ compilation.

Results. By applying the optimizations, the time in compiling IR to C++ has increased by a modest margin across all applications. As this phase is relatively short, the increased time is acceptable. As the outcome of the optimizations, we observe a consistent ∼15% lines of code fewer in generated C++ code. With less code generated, the time in compiling C++ code to executables has decreased by again a modest margin for all applications. As for the execution time, the optimized applications show a reduction of 9–37%, except for `true` whose running time is too short. To conclude, with

small investments in compilation time, we obtain perceivable performance benefits by deploying these optimizations.

F. Discussion

Threats to Validity. We strived for making the comparison with KLEE as fair as possible, *e.g.*, by implementing a similar random-path strategy and solver optimizations. However, it is possible that KLEE can be further tuned to perform better. We are aware that [57] reports overwhelming solver-time proportions in KLEE, which we only observe in a few cases in RQ2. This is potentially due to using different solvers (STP vs. Z3), different search strategies (DFS vs. random path), and changes made in KLEE over the last decade.

For SE tasks where "execution time" are already negligible or only take a small fraction of the overall time, compilation appears to be not economic in terms of end-to-end time, although the "execution speedup" is real.

Summary. Overall we find that GENSYM outperforms state-of-the-art SE tools. We observe average speedups of 4.6× in execution time vs. KLEE, and end-to-end speedups up to 9.4× in parallel vs. sequential execution. Their combination yields more than an order of magnitude speedup compared to KLEE. The current bottleneck of GENSYM lies in its current solver optimization implementation, which can be improved by adopting existing solutions. It would be also interesting to explore compiling to KLEE's backend, *e.g.*, reusing its state-of-the-art solver optimization implementation.

VIII. Related Work

Execution in Symbolic Analysis. Two closely related works are KLEE [32] and its predecessor EXE [39]. Both GENSYM and KLEE have similar functionality and the same input language. The difference lies in interpretation vs. compilation. Unlike KLEE's interpretation mode, EXE uses CIL [58] for source-to-source translation to instrument C programs. For multipath SE, EXE invokes the `fork` syscall to spawn OS processes coordinating with a search server process. We consider this approach heavyweight, since it delegates "control" to the OS and relies on OS-level concurrency and inter-process communication. In contrast, GENSYM compiles to continuations for a more lightweight concurrency implementation.

Several recent works on concolic execution propose using off-the-shelf compilation frameworks for IR-level instrumentation. SymCC [23] uses LLVM IR as the source and target of instrumentation, where the result will be compiled to native code by LLVM. SymQEMU [24] applies the similar idea to QEMU's IR. Under our view of "embedding" (Section II), instrumentation for concolic execution compiles to a shallow embedding of states and control. Compared to simple instrumentation that rarely alters control representations, GENSYM implements a more sophisticated compilation that transforms both value, state, and control representations.

Many other SE engines are also implemented as interpreters, *e.g.*, Symbolic Path Finder [59] for the JVM bytecode, Angr [60] for the VEX IR [61], etc. Our techniques can be applied to these languages/IRs as well.

Parallel Symbolic Execution. Cloud9 [33] is an extension of KLEE for use in distributed clusters. Cloud9 also uses a cooperative scheduler to simulate POSIX multithreading, but it is not based on CPS. Our approach would provide a unified way to symbolically executing multi-threaded programs as well, since we can use CPS-based cooperative scheduling to simulate OS preemptive scheduling, which is a future work.

Staats and Păsăreanu [50] design a client-server framework for parallelizing the JFP symbolic interpreter with a static path partition technique. Similar to GENSYM, Nowack et al. [51] propose a multi-thread architecture for parallel SE. However they use an interpreter for each worker, amplifying the interpretation overhead.

Partial Evaluation for Program Analysis. The idea of partially evaluating a static analyzer with respect to the input program is known as abstract compilation [62], which has been applied to different kinds of program analyses, *e.g.*, control-flow analysis [63], constraint-based analysis [64], abstract interpretation [65], and other variants of symbolic execution [66]. The underlying methodology of this paper is also aligned with abstract compilation.

GENSYM's approach is inspired by LLSC [20], [31] that uses multi-stage programming [15], [47] to achieve partial evaluation of symbolic interpreters. The novel contribution of GENSYM is generating the code in CPS form, enabling scalable cooperative parallelism and flexible support of heuristics. Compared to LLSC, GENSYM is much more mature in the sense that it supports more LLVM IR instructions, solver optimizations, file system models, and provides better performance and integration with coverage measuring tools.

Soufflé [67] uses the same partial evaluation idea as our work to specialize Datalog programs, mainly targeting rule-based program analysis. Both Soufflé and GENSYM can be considered generators for program analyzers that are specialized to specific input programs. Rojas and Păsăreanu [68] use partial evaluation to improve compositional SE, where a "path-specialized" program is used together with summaries of path conditions.

IX. Conclusion

We have presented the design and implementation of GENSYM. The novel insight of compiling SE to continuation-passing style with partially-evaluated symbolic interpreters also leads to practical performance speedups compared to state-of-the-art tools. We believe GENSYM's underlying techniques and methodology are key enablers for developing performant symbolic execution engines in the future.

Acknowledgement

We would like to thank the anonymous reviewers of ICSE '23 and ESEC/FSE '22 for their valuable feedback. We thank Xiangyu Zhang and Colin Gordon for their comments on the earlier drafts. This work was supported in part by NSF awards 1553471, 1564207, 1918483, 1910216, DOE award DE-SC0018050, as well as gifts from Facebook, Google, Microsoft, and VMware.

REFERENCES

[1] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT - a formal system for testing and debugging programs by symbolic execution," in *Reliable Software*. ACM, 1975, pp. 234–245.

[2] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[3] W. E. Howden, "Symbolic testing and the DISSECT symbolic evaluation system," *IEEE Trans. Software Eng.*, vol. 3, no. 4, pp. 266–278, 1977.

[4] L. A. Clarke, "A program testing system," in *ACM Annual Conference*. ACM, 1976, pp. 488–491.

[5] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of satisfiability*. IOS Press, 2021, pp. 1267–1329.

[6] C. Cadar and D. R. Engler, "Execution generated test cases: How to make systems code crash itself," in *SPIN*, ser. Lecture Notes in Computer Science, vol. 3639. Springer, 2005, pp. 2–23.

[7] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ESEC/SIGSOFT FSE*. ACM, 2005, pp. 263–272.

[8] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI*. ACM, 2005, pp. 213–223.

[9] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 50:1–50:39, 2018.

[10] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.

[11] C. Cadar, "Targeted program transformations for symbolic execution," in *ESEC/SIGSOFT FSE*. ACM, 2015, pp. 906–909.

[12] J. Chen, W. Hu, L. Zhang, D. Hao, S. Khurshid, and L. Zhang, "Learning to accelerate symbolic execution via code transformation," in *ECOOP*, ser. LIPIcs, vol. 109. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 6:1–6:27.

[13] S. Dong, O. Olivo, L. Zhang, and S. Khurshid, "Studying the influence of standard compiler optimizations on symbolic execution," in *ISSRE*. IEEE Computer Society, 2015, pp. 205–215.

[14] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*, ser. Prentice Hall international series in computer science. Prentice Hall, 1993.

[15] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs," in *GPCE*. ACM, 2010, pp. 127–136.

[16] O. Kiselyov, "Reconciling abstraction with high performance: A metaocaml approach," *Foundations and Trends® in Programming Languages*, vol. 5, no. 1, pp. 1–101, 2018.

[17] T. Rompf, K. J. Brown, H. Lee, A. K. Sujeeth, M. Jonnalagedda, N. Amin, G. Ofenbeck, A. Stojanov, Y. Klonatos, M. Dashti, C. Koch, M. Püschel, and K. Olukotun, "Go Meta! A case for generative programming and DSLs in performance critical systems," in *SNAPL*, ser. LIPIcs, vol. 32. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 238–261.

[18] P. Hudak, "Building domain-specific embedded languages," *ACM Comput. Surv.*, vol. 28, no. 4es, p. 196, 1996.

[19] C. Elliott, S. Finne, and O. de Moor, "Compiling embedded languages," *J. Funct. Program.*, vol. 13, no. 3, pp. 455–481, 2003.

[20] G. Wei, O. Bračevac, S. Tan, and T. Rompf, "Compiling symbolic execution with staging and algebraic effects," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 164:1–164:33, 2020.

[21] Y. Futamura, "Partial evaluation of computation process-an approach to a compiler-compiler," *Systems, Computers, Controls*, vol. 25, pp. 45–50, 1971.

[22] ——, "Partial evaluation of computation process - an approach to a compiler-compiler," *High. Order Symb. Comput.*, vol. 12, no. 4, pp. 381–391, 1999.

[23] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!" in *USENIX Security Symposium*. USENIX Association, 2020, pp. 181–198.

[24] ——, "SymQEMU: Compilation-based symbolic execution for binaries," in *NDSS*. The Internet Society, 2021.

[25] G. J. Sussman and G. L. Steele Jr, "Scheme: An interpreter for extended lambda calculus," *MIT AI Memo*, 1975.

[26] A. W. Appel, *Compiling with Continuations*. Cambridge University Press, 1992.

[27] G. L. Steele, "Rabbit: A compiler for scheme," USA, Tech. Rep., 1978.

[28] A. Kennedy, "Compiling with continuations, continued," in *ICFP*. ACM, 2007, pp. 177–190.

[29] A. W. Appel and D. B. MacQueen, "Standard ML of New Jersey," in *PLILP*, ser. Lecture Notes in Computer Science, vol. 528. Springer, 1991, pp. 1–13.

[30] Y. Cong, L. Osvald, G. M. Essertel, and T. Rompf, "Compiling with continuations, or without? whatever," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, pp. 79:1–79:28, 2019.

[31] G. Wei, S. Tan, O. Bračevac, and T. Rompf, "LLSC: A parallel symbolic execution compiler for LLVM IR," in *ESEC/SIGSOFT FSE*. ACM, 2021, pp. 1495–1499.

[32] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*. USENIX Association, 2008, pp. 209–224.

[33] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *EuroSys*. ACM, 2011, pp. 183–198.

[34] The Free Software Foundation, "GNU core utilities," https://www.gnu.org/software/coreutils/, 2022, accessed: 2022-08-10.

[35] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*. ACM, 1977, pp. 238–252.

[36] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010, pp. 317–331.

[37] J. C. Reynolds, *User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction*. New York, NY: Springer New York, 1978, pp. 309–317.

[38] J. Gibbons and N. Wu, "Folding domain-specific languages: Deep and shallow embeddings (functional pearl)," in *ICFP*. ACM, 2014, pp. 339–347.

[39] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," in *CCS*. ACM, 2006, pp. 322–335.

[40] KLEE Team, "KLEE's version of uClibc ," https://github.com/klee/klee-uclibc, 2022, accessed: 2022-08-17.

[41] U. Jørring and W. L. Scherlis, "Compilers and staging transformations," in *POPL*. ACM Press, 1986, pp. 86–96.

[42] S. Thibault, C. Consel, J. L. Lawall, R. Marlet, and G. Muller, "Static and dynamic program compilation by interpreter specialization," *High. Order Symb. Comput.*, vol. 13, no. 3, pp. 161–178, 2000.

[43] M. Sperber and P. Thiemann, "Realistic compilation by partial evaluation," in *PLDI*. ACM, 1996, pp. 206–214.

[44] R. Y. Tahboub, G. M. Essertel, and T. Rompf, "How to architect a query compiler, revisited," in *SIGMOD Conference*. ACM, 2018, pp. 307–322.

[45] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer, "Practical partial evaluation for high-performance dynamic language runtimes," in *PLDI*. ACM, 2017, pp. 662–676.

[46] W. Taha and T. Sheard, "Multi-stage programming with explicit annotations," in *PEPM*. ACM, 1997, pp. 203–217.

[47] ——, "MetaML and multi-stage programming with explicit annotations," *Theor. Comput. Sci.*, vol. 248, no. 1-2, pp. 211–242, 2000.

[48] N. D. Jones, "Transformation by interpreter specialisation," *Sci. Comput. Program.*, vol. 52, pp. 307–339, 2004.

[49] J. P. B. Puente, "Persistence for the masses: RRB-vectors in a systems language," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, pp. 16:1–16:28, 2017.

[50] M. Staats and C. S. Păsăreanu, "Parallel symbolic execution for structural test generation," in *ISSTA*. ACM, 2010, pp. 183–194.

[51] M. Nowack, K. Tietze, and C. Fetzer, "Parallel symbolic execution: Merging in-flight requests," in *Haifa Verification Conference*, ser. Lecture Notes in Computer Science, vol. 9434. Springer, 2015, pp. 120–135.

[52] LLVM Developers, "LLVM passes - promote memory to register," https://llvm.org/docs/Passes.html#mem2reg-promote-memory-to-register, 2022, accessed: 2022-08-10.

[53] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *PLDI*. ACM, 2012, pp. 193–204.

[54] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 337–340.

[55] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *CAV*, ser. Lecture Notes in Computer Science, vol. 4590. Springer, 2007, pp. 519–531.
[56] KLEE Team, "KLEE's POSIX Simulation Model," https://github.com/klee/klee/tree/master/runtime/POSIX, 2022, accessed: 2022-08-17.
[57] H. Palikareva and C. Cadar, "Multi-solver support in symbolic execution," in *CAV*, ser. Lecture Notes in Computer Science, vol. 8044. Springer, 2013, pp. 53–68.
[58] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: intermediate language and tools for analysis and transformation of C programs," in *CC*, ser. Lecture Notes in Computer Science, vol. 2304. Springer, 2002, pp. 213–228.
[59] C. S. Păsăreanu and N. Rungta, "Symbolic PathFinder: symbolic execution of Java bytecode," in *ASE*. ACM, 2010, pp. 179–180.
[60] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2016, pp. 138–157.
[61] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *PLDI*. ACM, 2007, pp. 89–100.
[62] D. Boucher and M. Feeley, "Abstract compilation: A new implementation paradigm for static analysis," in *CC*, ser. Lecture Notes in Computer Science, vol. 1060. Springer, 1996, pp. 192–207.
[63] J. M. Ashley and R. K. Dybvig, "A practical and flexible flow analysis for higher-order languages," *ACM Transactions on Programming Languages and Systems*, vol. 20, no. 4, pp. 845–868, 1998.
[64] T. Amtoft, "Partial evaluation for constraint-based program analyses," *BRICS Report Series*, vol. 6, no. 45, 1999.
[65] G. Wei, Y. Chen, and T. Rompf, "Staged abstract interpreters: Fast and modular whole-program analysis via meta-programming," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 126:1–126:32, 2019.
[66] S. Tan, G. Wei, and T. Rompf, "Towards partially evaluating symbolic interpreters for all," in *PEPM*. ACM, 2022.
[67] H. Jordan, B. Scholz, and P. Subotic, "Soufflé: On synthesis of program analyzers," in *CAV (2)*, ser. Lecture Notes in Computer Science, vol. 9780. Springer, 2016, pp. 422–430.
[68] J. M. Rojas and C. S. Păsăreanu, "Compositional symbolic execution through program specialization," *BYTECODE'13 (ETAPS)*, 2013.