Towards Partially Evaluating Symbolic Interpreters for All (Short Paper)

Shangyin Tan
tan279@purdue.edu
Purdue University
West Lafayette, IN, USA

Guannan Wei
guannanwei@purdue.edu
Purdue University
West Lafayette, IN, USA

Tiark Rompf
tiark@purdue.edu
Purdue University
West Lafayette, IN, USA

Abstract

Symbolic execution is a program analysis technique to automatically explore the execution space of programs by treating some inputs symbolically. To efficiently perform symbolic execution, one emerging way is to construct a compiler that translates input programs to symbolic programs without the interpretation overhead. Previous work has explored compiling nondeterministic symbolic execution by partially evaluating a symbolic interpreter.

In this paper, we follow a "semantics-first" approach and investigate compiling concolic execution and backward symbolic execution by multi-stage programming. In particular, we construct variants of staged symbolic interpreters that can be partially evaluated using the Lightweight Modular Staging (LMS) framework. We demonstrate our approach using a simple low-level intermediate representation (IR) and evaluate the prototype implementations for the LLVM IR. Our concolic compiler shows comparable performance to SymCC, a state-of-the-art concolic compiler, and our backward symbolic compiler solves tasks that are difficult for forward execution engines. The demonstrated approach shows a unifying methodology that can be applied to compiling diverse flavors of symbolic execution.

CCS Concepts: • **Software and its engineering** → **Interpreters**; **Source code generation**; **Automated static analysis**; **Software testing and debugging**.

Keywords: symbolic execution, code generation, compilers, interpreters

1 Introduction

Symbolic execution is a program analysis technique widely that has been used in testing, analysis, and verification [2, 21]. The key idea of symbolic execution is to execute a program with symbolic inputs and to collect path conditions over these symbolic inputs. Then, automated theorem provers (e.g. SMT solvers) can be used to solve the path conditions and generate inputs satisfying these conditions.

However, applying pure symbolic execution that nondeterministically explores all branches is intractable on large real-world programs. This problem is known as the path explosion problem. To effectively scale up symbolic execution, researchers have proposed different flavors or strategies of symbolic execution. To name a few examples, concolic execution [3, 17, 32], a popular variant widely used in computer security research, mixes concrete and symbolic execution and only collects conditions of paths guided by the concrete input. Fork-based nondeterministic symbolic execution [8, 10] maintains multiple execution states and often combines with search heuristics. In addition, backward symbolic execution [9, 24, 27], a more exotic flavor, starts from a given program point and finds inputs leading to this specific point, avoiding exploring unrelated execution space.

Orthogonal to the effort to conquer the intrinsic complexity of symbolic execution due to path explosion, this paper concerns the accidental complexity of implementing *efficient* engines for symbolic execution, which can be non-trivial in practice as well. Building symbolic execution engines by interpreters (e.g. KLEE [8]) is a popular but inefficient approach due to the well-known interpretation overhead. Instrumentation-based execution engines deliver higher performance, but it is not straightforward to instrument symbolic execution other than the single-path forward execution semantics. As a promising new direction, using compiler techniques to implement symbolic execution has gained increasing attention recently: different compilation-based methods have been successfully applied to different flavors of symbolic execution. For example, our prior work [39, 41] uses staging to compile fork-based symbolic execution; SymCC [29] uses LLVM's transformation infrastructure to compile concolic execution. Nevertheless, it is unclear if those compilation techniques can be applied to symbolic execution variants other than their original consideration.

In this paper, we argue that principled metaprogramming is feasible and effective to construct diverse, if not all, flavors of symbolic execution compilers. To demonstrate our approach, we extend our previous staging-based "semantics-first" approach [39] to compiling *concolic* execution and *backward* symbolic execution. By embedding staged concolic and backward symbolic interpreters in Scala and LMS [31], the compilation and code generation are performed by partially evaluating the symbolic interpreters, following the 1st Futamura projection [13, 14]. We demonstrate a concolic interpreter by modularly composing the more primitive concrete

PEPM 2022, Jan 17-18, 2022, Philadelphia, PA, USA
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

and symbolic interpreters. We also sketch the implementation of the backward execution compiler.

The performance of the derived symbolic compilers on LLVM IR is examined using a set of small programs. We compare the derived concolic compiler with SymCC [29], a concolic execution compiler using LLVM's compilation framework to perform instrumentation, showing similar speedups. The effectiveness of the backward symbolic execution compiler is evaluated with programs that are difficult for forward symbolic engines. Our prototype backward compiler compiles the benchmarks to backward executing programs that yield the expected inputs to reach the target locations.

Contributions.

- We discuss a unified methodology for constructing compilers for symbolic execution (§ 2) based on the 1st Futamura projection, which generalizes previous works and derives novel symbolic execution compilers.
- After briefly introducing the language for presentation (§ 3), we show how to incrementally build the concolic interpreter and add staging annotations to obtain the concolic compiler (§ 4). We discuss how to apply the same approach to backward symbolic execution (§ 5).
- We empirically evaluate the performance and correctness of our prototype compilers implemented on LLVM (§ 6).

Finally, we discuss related work (§ 7), conclude the paper, and discuss future directions (§ 8).

2 The Essence of Compiling Symbolic Execution vis Staging

The essence of our approach to compiling symbolic execution is first to build a symbolic interpreter and then specialize it with repsect to the given input program using multi-stage programming. This section reviews the key concepts of staging using LMS, draws the connection between program specialization and symbolic execution compilers, and gives an overview of our implementation architecture.

2.1 Interpreter Specialization via Staging with LMS

Metaprogramming is the programming technique of writing programs to manipulate and generate other programs. One of the powerful metaprogramming techniques is multi-stage programming (MSP) [36, 37], which has been available as a language feature [4, 22, 33, 34] or a framework [7, 28, 31]. The key idea behind MSP (staging for short) is that program execution can be split into stages by the frequency of execution or availability of inputs. Earlier stages generate code *specialized* to the known arguments, which can be executed later when unknown arguments become available [20, 23]. This process provides a reliable way to realize partial evaluation [19] controlled by the programmer. The partially-evaluated (i.e. specialized) program usually runs faster since the computations over known inputs have been performed statically.

Our implementation uses the Lightweight Modular Staging framework (LMS) [31] for staging. LMS utilizes type-level annotations to specify binding times for next-stage expressions, i.e. `Rep[T]` is the type of next stage values of `T`. To see an example of staging with LMS, let us consider the following *unstaged* interpreter for arithmetic expressions. The interpreter `eval` takes the abstract syntax tree (AST) of expressions (`Expr`) and an environment σ and evaluates the expression program to integers:

```
def eval(e: Expr, σ: Map[String,Int]): Int =
  e match {
    case Const(i) ⇒ i
    case Var(x) ⇒ σ(x)
    case Add(e1, e2) ⇒ eval(e1, σ) + eval(e2, σ)
    ...
  }
```

The *staged* version of this interpreter adds the annotation `Rep` to the environment type and the result type, indicating that they are values known at a later stage:

```
def eval(e: Expr, σ: Rep[Map[String,Int]]): Rep[Int]
  = e match {
    case Const(i) ⇒ i
    case Var(x) ⇒ σ(x)
    case Add(e1, e2) ⇒ eval(e1, σ) + eval(e2, σ)
    ...
  }
```

In LMS, operations over expressions of `Rep[_]` types (e.g. store lookup `σ(x)`) are overloaded implicitly, redirecting to methods that construct staged code. Therefore, the staged `eval` can reuse the same piece of code from its unstaged counterpart.

The staged `eval` function can be specialized to a statically known program AST. For example, specializing `eval` to program p = `Add(Const(3), Var("x"))` unfolds the recursive function calls of `eval`, eliminates the static computations over AST, and generates code that is equivalent to:

```
def eval_p(σ: Map[String,Int]): Int = 3 + σ("x")
```

The specialized code does not depend on any static program AST. Therefore, we can view it as the result of compilation. This process of specializing interpreters is known as the first projection of Futamura [13, 14].

By writing interpreters with multi-stage programming facilities, we can observe that *a staged interpreter is essentially a compiler*. Naturally, if we build staged symbolic interpreters, they can be used as symbolic execution compilers [39]. Following this approach, the construction of symbolic execution compilers boils down to two simple steps: (1) express the symbolic execution semantics in the form of an interpreter, (2) analyze the interpreter program and add suitable staging annotations. The rest of this paper will discuss how to apply this idea to concolic and backward symbolic executions.

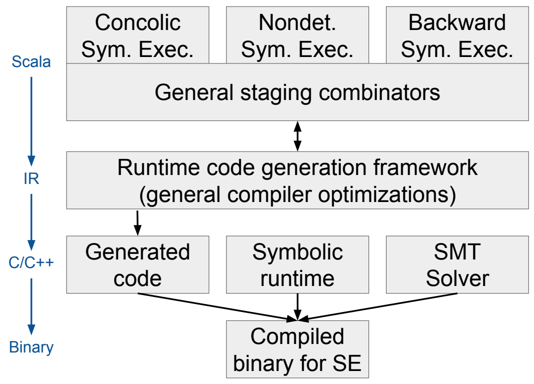

Figure 1. The architecture of our approach.

2.2 Architecture of Our Approach

With the 1st Futamura projection and interpreter specialization in mind, our task to architect symbolic compilers is divided into two rather independent ones: (1) the development of symbolic interpreters and (2) the infrastructure to support this development, including the auxiliary functions for code generation, runtime symbolic state representations, SMT solver libraries, etc. The infrastructure should provide enough abstractions and facilities to implement the diverse variants of symbolic execution.

Figure 1 shows the architecture of our approach. The middle part of the architecture is the runtime code generation framework that provides an extensible intermediate representation (IR), general optimizations, and code generation. We use the LMS [31] framework written in Scala for this purpose. Built on top of this layer, we develop a staging combinator library that abstracts over the operations manipulating staged program states (e.g. stack, heap, and path conditions) and the construction of staged symbolic expressions. This staging combinator library is shared among the staged symbolic interpreters, which can vary according to the desired symbolic semantics, as shown at the top of Figure 1. Given an input program, the staged symbolic interpreter is specialized when invoking the staging combinators, where the residual program is first represented by the IR of LMS. After optimizing over the IR inside LMS, the residual program for symbolic execution will be generated as C++ code. Combined with the symbolic runtime and the SMT solver library, the generated C++ code is further compiled to an executable (e.g. by clang). Running this executable faithfully performs the symbolic semantics defined by the top-level symbolic interpreter. With the interpretation overhead eliminated and the translation from Scala to C++, the executables usually could run an order-of-magnitude faster.

3 Preliminaries

We use a simplified static single assignment form (SSA) language SIR modeled after the LLVM IR to demonstrate our approach. Similar to the LLVM IR, SIR is a language-agnostic representation, and the approach can be readily applied to a large portion of LLVM IR (§ 6).

$$
\begin{array}{llll}
v \in \mathsf{Atom} & ::= x \mid i32 \quad x \in \mathsf{Var} \quad Op_2 \in \{+,-,\times,=,\neq,\dots\} \\
i \in \mathsf{Inst} & ::= \mathsf{Assign}(x, v_i) \mid \mathsf{Store}(v_1, v_2) \mid \mathsf{CondBr}(v, l_1, l_2) \\
 & \quad \mid \mathsf{Jmp}(l) \mid \mathsf{Return}(v) \\
v_i \in \mathsf{ValInst} & ::= Op_2(v_1, v_2) \mid \mathsf{Load}(v) \mid \mathsf{Alloca}(v) \\
b \in \mathsf{Block} & ::= \mathsf{block}(l, \{\, i, \dots \}) \quad p \in \mathsf{Prog} ::= b, \dots
\end{array}
$$

Figure 2. The abstract syntax of SIR.

The syntax of SIR is shown in Figure 2. A program p is a sequence of basic blocks. A basic Block has a label as its identifier and a list of instructions, whose last instruction is either a return statement or jumps to another block. The language defines a few instructions manipulating the memory, which all have standard forms and semantics: Alloca allocates a sized area; Load retrieves the value in a memory cell; Store puts a value into a memory cell; Assign binds the result of a value instruction to a local name. The language only has 32-bit unsigned integers, which model both values and memory locations. We also define a set of standard arithmetic operations Op_2. Function definitions and calls are extended in § 5 to demonstrate backward symbolic execution.

Concrete Semantics. Programs in SIR have standard concrete semantics, resembling the similar behavior as in LLVM's IR. We illustrate a monadic semantics where the underlying state monad carries the program `State`, consisting of the heap and stack memory. In the following snippet, we provide the type signatures of the monadic interpreters and omit the actual implementation.

```
type State = (Heap, Stack)
type M[T] = StateM[State, T]
type Value = Int32
def evalAtom_c(v: Atom): M[Value]
def evalValInst_c(vi: ValInst): M[Value]
def evalInst_c(i: Inst): M[Value]
```

The defined interpreters such as `evalAtom` and `evalInst` handle different syntactic categories defined in Figure 2. Their implementations (omitted) in Scala use `for`-comprehension syntax for moandic sequential bindings, and `ret: T ⇒ M[T]` for the monadic "unit" function.

4 Compiling Concolic Execution

The idea of concolic execution is to perform symbolic execution on the paths guided by concrete values. In addition to the concrete state, concolic execution engine also mantains a symbolic state during the execution that tracks all symbolic values and expressions. The execution engine tracks the symbolic condition of the concrete path, which can be used to generate new inputs.

```
def findBlock: Label ⇒ List[Inst] = ...
def evalAtom_s(a: Atom): M[Option[Sym]] = ...
def evalValInst_s(v: ValInst): M[Option[Sym]] = ...
def evalInst_s(i: Inst): M[Option[Sym]] = i match {
  case Assign(x, v) ⇒ for {
      sv ⟵ evalValInst_s(v)
      _  ⟵ updateSymEnv(x, sv)
    } yield sv
  case Return(v) ⇒ evalAtom_s(v)
  case Store(v1, v2) ⇒ for {
      addr ⟵ evalAtom_c(v1)
      sv   ⟵ evalAtom_s(v2)
      _    ⟵ updateSymMem(addr, sv)
    } yield sv
  case _ ⇒ ret(None) // control inst. not handled
}
```

Figure 3. Symbolic interpreters (excerpted) for SIR

Once we obtain the condition of a single path, there are multiple ways to generate new inputs. For example, Godefroid et al. [18] propose to negate each condition in the constraint set after a complete run of the program. Their approach assigns a bound number to each input to ensure that the newly generated inputs can reach distinct paths in the program. Another realistic and powerful way to use concolic execution is to combine it with fuzz testing [25]. With mutation strategies, the fuzzer can provide numerous concrete inputs to guide the concolic execution and produce more inputs leading to the unexplored space. Nevertheless, the strategy of using the path condition to generate inputs is mostly orthogonal to the concolic execution itself and is not our focus in this paper.

In this section, we focus on the semantics and interpreter for concolic execution and explain the infrastructure to obtain a compiler for concolic execution.

4.1 Concolic Semantics and Interpreter

The concolic execution semantics can be derived from the concrete semantics. First, in addition to concrete states, the concolic semantics needs to maintain symbolic states (i.e. heap/stack that maps to symbolic expressions) and simultaneously performs concrete and symbolic computation. Second, the concolic semantics collects the symbolic path conditions following the concrete path, which happens when branching. Based on these observations, we use combinators to compose the concrete interpreter and the symbolic interpreter, meanwhile adapting the execution of control instructions. We reuse the concrete interpreters from § 3 and describe the new symbolic interpreter next.

Symbolic Interpreter. The excerpted symbolic interpreters are shown in Figure 3. We first introduce symbolic values `Sym` in addition to the concrete values, which can be either

```
val evalInst_conc = Δ_fix(evalInst_c, evalInst_s, {
  base ⇒ rec ⇒
  case CondBr(c, l1, l2) ⇒ for {
      (cc, sc) ⟵ evalAtom_conc(c)
      _    ⟵ updatePC(cc, sc)
      res ⟵ if (cc) forM(findBlock(l1))(rec)
             else forM(findBlock(l2))(rec)
    } yield res
  case Jmp(l) ⇒ forM(findBlock(l))(rec)
  case inst ⇒ base(inst)
})
```

Figure 4. The combined symbolic interpreter (excerpted) for SIR

a symbol or a symbolic expression. A symbolic expression consists of an operator and operands. The state carried by the state monad `M` now additionally includes the symbolic heap and stack. During concolic execution, since not all values have a symbolic meaning the symbolic interpreters return an `Option[Sym]` value (wrapped in the state monad), expressing this partiality. The lookup operations over symbolic memory are undefined (`None`) if there is no symbolic value for the corresponding location.

The symbolic interpreter is similar to its concrete counterpart in many aspects. For example, when evaluating the Store instruction symbolically in $\texttt{evalInst}_s$, we first evaluate the address *concretely*, as we disallow symbolic locations due to adopting a simpler memory model. Then, we evaluate the value to be stored symbolically, which will be used to update the symbolic memory. Since the symbolic evaluation does not guarantee success, `updateSymMem` only updates the symbolic memory if `sv` is not `None`. The difference compared with the concrete evaluation of Store instructions is that we modify the symbolic state with a possibly symbolic value.

The symbolic interpreter yields `None` for a few other instructions. For example, we do not handle the `CondBr` case in $\texttt{evalInst}_s$. Later, these cases will be overridden in the composed concolic interpreter, where the concrete evaluation provides the selected branch.

Combining Symbolic and Concrete Interpreter. The symbolic interpreters are modularly defined, which then can be composed with the concrete interpreter. To obtain the composed concolic interpreter, we first define several auxiliary combinators. Given the similarity of the symbolic and concrete interpreter that they both operate on the program IR, it suggests using the Δ combinator that lifts two (monadic) functions with a common source type to a single function that returns a pair of results:

```
def Δ(f: A ⇒ M[B], g: A ⇒ M[C]): A ⇒ M[(B, C)]
```

However, this is not enough to obtain the concolic interpreter: (1) it does not provide a chance to override the execution for `CondBr`; (2) the concolic interpreter needs to recurs over the subsequent instructions, but this Δ combinator does not provide a way to refer the combined interpreter itself.

Therefore, we combine Δ and the idea of (call-by-value) fixed-point combinator to derive the Δ_{fix} combinator:

```
def Δ_fix(f: A ⇒ M[B], g: A ⇒ M[C],
  ev: (A ⇒ M[(B, C)]) ⇒ (A ⇒ M[(B, C)]) ⇒
      A ⇒ M[(B, C)]): A ⇒ M[(B, C)] =
  a ⇒ ev(f Δ g)(Δ_fix(f, g, ev))(a)
```

The additional third argument `ev` is a higher-order function that takes two functions of type `A ⇒ M[(B, C)]` and the ordinary argument of type `A`. The first function provides the non-recursive base case, i.e. `f Δ g`. And the second function is the function referring `ev` itself, i.e., the composed function that can handle potentially new cases. With Δ_{fix} in hand, we can now define the concolic interpreter (Figure 4). If the instruction has no control effect, simply combining the concrete and symbolic interpreter is sufficient (by calling `base`). However, if the instruction potentially changes the control flow, we ignore the previous defined two interpreters and provide a new interpretation. For example, in the case of `CondBr`, we obtain a concrete condition `cc` and an optional symbolic condition `sc` by calling $\texttt{evalAtom}_{conc}$, which is obtained using Δ as well. The eventual result of a `CondBr` case depends on the concrete condition `cc`. In addition, we collect the symbolic path condition by calling `updatePC`. To execute the instructions in another block in the case of `CondBr` and `Jmp`, we apply the auxiliary function `findBlock` to the given label, and `forM` invokes the self-recursive monadic function `rec` with the last result returned.

Now, the interpreter $\texttt{evalInst}_{conc}$ correctly expresses the semantics of concolic execution. Next, we add staging annotations to it to obtain the concolic execution compiler.

4.2 Staged Concolic Interpreter

In our framework, we convert the interpreter to a staged interpreter by adding binding-time annotations at the type level. With suitable implicit overloading functions, the main code of the interpreter can remain the same. We first change the state type underlying the monad to be the *staged* state using the `Rep` annotation. Therefore, the type of the concolic state monad becomes:

```
  type M[T] = StateM[Rep[(CState, SState)], T]
```

where `CState` and `SState` is the concrete and symbolic state, respectively. The auxiliary functions (e.g. `updateSymEnv` and `updatePC`) that manipulate the underlying state representation are also adapted to operate over staged states. When invoked, these operations construct the next-stage code representation using the LMS framework.

4.3 Symbolic Backend

The generated code dynamically constructs symbolic expressions; thus the backend needs to support a wide range of symbolic operations, which eventually constructs SMT expressions. Symbolic expressions in the backend have two forms: they are either symbolic variables like $\mathsf{Sym}(x)$ or symbolic expressions like $\mathsf{Sym}(op, x, ...)$. The backend representation of the symbolic memory maps identifiers or memory locations to symbolic expressions. This part of the backend is shared among other variants of symbolic execution.

5 Compiling Backward Symbolic Execution

Backward symbolic execution is a specialized flavor of symbolic execution that aims to find inputs that reach target program locations. By executing the program backward from the desired location to the entry point, the backward symbolic executor is less likely to take spurious paths that lead to irrelevant parts of the program.

In this section, we focus on compiling a specific backward symbolic execution variant called *call-chain backward symbolic execution* (CCBSE) [24]. CCBSE builds upon the forward nondeterministic symbolic execution and uses it as a subroutine. Nondeterministic symbolic execution, unlike concolic execution, mixes symbolic and concrete values in a single store. When evaluating a symbolic condition, the symbolic executor attempts to explore both paths nondeterministically if they are satisfiable. Our previous work [39] elaborates a semantic-first approach to compile nondeterministic symbolic execution, which expresses the choice of branches using algebraic effects and handlers. In the rest of this section, we also make use of our previous compiler construction of fork-based symbolic execution.

5.1 CCBSE Semantics

To demonstrate how CCBSE works, we make some small changes to the syntax of SIR. We extend ValInst with an additional call instruction. We also add function definitions FunDef to the language, which contains arguments and a list of blocks. A program consists of a list of FunDef.

CCBSE starts the execution from the function f that contains the target program location, treating all the arguments as symbolic ones. Then, CCBSE executes all the functions that call f using the fork-based symbolic execution. When reaching a call instruction to the function f, the executor tries to concatenate the paths from the current function that calls f. By performing this backward process recursively over the call-graph, CCBSE terminates when a path is found from the `main` function (i.e. from the entry point to the target location).

The challenge to compile CCBSE here is twofold: (1) CCBSE requires an additional runtime to schedule the execution of each function backward following the call-chain, (2) we need to properly define the concatenation of paths between functions for the symbolic interpreter.

5.2 Compiling CCBSE

To schedule the symbolic execution of each individual function, we need a representation of the call graph at runtime.

This can be readily achieved if we only consider direct function calls, which are statically represented in the AST. The call graph is calculated at staging time and stored into a next-stage structure. Then, we can develop backend facilities that rely on the next-stage call graph to schedule the symbolic executor according to the CCBSE semantics.

Composing paths from prior results is also crucial to build CCBSE correctly. After executing the target function f, a "summary" is associated with f, containing all feasible paths that start from f and reach the target location. In our framework, we encode the summary as a list of states and path conditions `List[(State, PC)]`, representing all possible execution results. This encoding directly comes from the result type of our previous nondeterministic symbolic executor [39]. Then, when the symbolic executor encounters function f later from other functions, instead of blindly executing function f again, the symbolic executor reuses the previous summary. In this way, we concatenate the current execution paths with the previous explored paths and obtain the desired CCBSE benefit.

Implementing the concatenation requires a small modification to the `call` case of the interpreter where the callee has a valid summary. Suppose the caller is g and the callee is f, we invoke the backend function `call-concat` in the generated code. In this function, we obtain the path constraint sets from f's summary, and for each path condition set in the summary, we add the constraint set to the current state of g and then call f. This call to f only explores one path associated with the specific constraint set added to g's current state. Finally, `call-concat` returns the list of valid paths that concatenate g and f, and this list later becomes part of g's summary.

Discussion. The implementation of CCBSE demonstrates the modular and flexible aspects of our approach. Instead of implementing the runtime support for concatenating paths and a stand-along scheduler, our approach enables smooth integration for more complex semantics. Depending on whether the change influences the operation semantics of symbolic execution, the modification can be made directly on the interpreter or on the backend code that uses the symbolic executor as a subroutine.

6 Evaluation

To evaluate our approach, we have implemented the concolic compiler and the backward compiler on a subset of LLVM, supporting more than 40 instructions. Both compilers have less than 700 LoC for the staged interpreters in Scala. The staged interpreters are adapted from our previous work [41] with less than 200 LoC changes, respectively. Our symbolic backend uses the STP theorem prover [15].

We demonstrate the performance of our approach by evaluating the derived compilers on synthetic programs and small realistic programs. All experiments are conducted on an Intel i7-8750H machine running Ubuntu 20.04 with 16 GB main memory. The running time numbers reported are the medians of 10 runs. The benchmarks and reproduction of the experiments are available online at https://shangyit.me/pepm-experiment.

Table 1. Evaluation result of concolic compilers. #inputs: number of generated inputs; T_{staging}: staging time of our tool; $T_{\text{symcc-comp}}$: SymCC compilation time; T_{conc}: running time of our generated code; T_{symcc}: running time of SymCC's generated code.

Benchmark	#inputs	T_{staging}	$T_{\text{symcc-comp}}$	T_{conc}	T_{symcc}
BinSearch	8	380.7 ms	304.5 ms	41.6 ms	49.3 ms
BubbleSort	7	113.1 ms	315.4 ms	47.95 ms	144.4 ms
Knapsack	22	832.3 ms	332.5 ms	168.9 ms	207.1 ms
QuickSort	28	500.8 ms	348.5 ms	123.8 ms	109.8 ms

Benchmarking Concolic Compiler. We compare our concolic LLVM compiler with SymCC, a concolic execution engine that utilizes LLVM to symbolically instrument code. The benchmarks we choose are small but realistic programs: BinSearch, BubbleSort, Knapsack, and QuickSort. We record the running times of a single execution in Table 1.

We run SymCC with its simple Z3 [12] backend, which has a similar implementation compared to ours. We also record the result of one complete run of the target program with the same input seed to ensure that the two engines generate the same set of inputs. Overall, we find our prototype produces symbolic programs that have comparable performance with respect to SymCC.

Benchmarking Backward Symbolic Compiler. To evaluate the performance of our CCBSE compiler, we deliberately use a few small (less than 40 LoC) synthetic benchmarks adapted from [24] that impose some difficulties for forward symbolic execution engines - these programs have intensive computations that do not eventually reach the target location, and forward symbolic execution engines are often stuck in those computations.

One of the benchmark programs (p_4) is shown in the appendix (Figure 5). The target function `f` has 2^4 paths induced by the `for` loop, and only one path leads to the error we want to trigger. Normal forward symbolic execution needs to execute 2^4 paths everytime function `f` is invoked. However, as CCBSE first explores 2^4 paths from the target function `f` and obtains a summary containing the erroneous path, each later invocation of function `f` only executes the path from the summary, which makes the overall execution more efficient.

We present the staging time and running time of each benchmark program in Table 2. We also compares the running time with our previous forward symbolic execution compiler LLSC [41], and the running time of the CCBSE compiler (T_{CCBSE}) is generally faster than the LLSC compiler (T_{LLSC}). The observed behavior concludes that our backward symbolic execution compiler produces efficient code.

Table 2. Experiment result of the CCBSE compiler.

	p1	p2	p3	p4
T_{staging}	195 ms	228 ms	205 ms	210 ms
T_{CCBSE}	116 ms	7 ms	11 ms	468 ms
T_{LLSC}	220 ms	26 ms	46 ms	13.9s

7 Related Work

Symbolic execution [2, 6, 21] is a program analysis technique widely used in testing, bug/crash finding, and verification. Many practical tools [8, 26, 29, 38] are developed based on symbolic execution. Concolic execution [25, 30, 42], as one of the popular variants, has been successfully used by the security community due to its scalability. Readers may find the idea behind symbolic execution similar to partial evaluation [19], as they both transform programs to forms that only depend on unknown inputs. This paper uses multi-stage programming [35, 36] and the LMS framework [31] to perform partial evaluation guided by annotations. The idea of multi-stage programs is closely related to multi-level binding-time analysis [16], although in this work we only exploit two stages of interpreters.

This paper is built upon our prior work [39, 41], which uses staging to compile fork-based symbolic execution. SymCC [29] is another closely related work using an LLVM pass to perform the instrumentation of concolic execution. The idea of using program specialization to accelerate program analysis can be found in other analyses [1, 5, 11, 40] too.

8 Conclusion and Future Work

This short paper demonstrates our work towards a general methodology of constructing symbolic execution compilers of diverse flavors based on staging. We present two new variants of symbolic compilers following this approach, i.e., the concolic and backward symbolic execution compiler. The preliminary evaluation result shows that our symbolic compilers can generate performant and effective compiled symbolic programs, meanwhile allowing the developers to use a high-level productive language to realize this goal.

Our exploration and demonstration are by no means exhaustive. It would be interesting to see how this approach can be applied to more variants with different language features, e.g., backward execution for higher-order languages [27]. Another interesting direction is to combine the compiler with a fuzzer [25, 42]. We expect a synergy to combine concolic execution compilers with fuzzers that are also specialized to the input grammar structure.

References

[1] Gianluca Amato and Fausto Spoto. 2001. Abstract Compilation for Sharing Analysis. In *Functional and Logic Programming, 5th International Symposium, FLOPS 2001, Tokyo, Japan, March 7-9, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2024)*, Herbert Kuchen and Kazunori Ueda (Eds.). Springer, 311–325. https://doi.org/10.1007/3-540-44716-4_20

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3 (2018), 50:1–50:39. https://doi.org/10.1145/3182657

[3] Thomas Ball and Jakub Daniel. 2015. Deconstructing Dynamic Symbolic Execution. In *Dependable Software Systems Engineering*. NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 40. IOS Press, 26–41.

[4] Alan Bawden. 1999. Quasiquotation in Lisp. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA, January 22-23, 1999. Technical report BRICS-NS-99-1*, Olivier Danvy (Ed.). University of Aarhus, 4–12.

[5] Dominique Boucher and Marc Feeley. 1996. Abstract Compilation: A New Implementation Paradigm for Static Analysis. In *Proceedings of the 6th International Conference on Compiler Construction (CC '96)*. Springer-Verlag, London, UK, UK, 192–207. http://dl.acm.org/citation.cfm?id=647473.727587

[6] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT - a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software* (Los Angeles, California). ACM, New York, NY, USA, 234–245. https://doi.org/10.1145/800027.808445

[7] Ajay Brahmakshatriya and Saman P. Amarasinghe. 2021. BuildIt: A Type-Based Multi-stage Programming Framework for Code Generation in C++. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 39–51. https://doi.org/10.1109/CGO51591.2021.9370333

[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* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[9] Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: a powerful approach to weakest preconditions. In *PLDI*. ACM, 363–374.

[10] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: a platform for in-vivo multi-path analysis of software systems. In *ASPLOS*. ACM, 265–278.

[11] Daniel Damian. 1999. Partial evaluation for program analysis. *Progress report, BRICS PhD School, University of Aarhus* (1999).

[12] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[13] Yoshihiko Futamura. 1971. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems, Computers, Controls* 25 (1971), 45–50.

[14] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (01 Dec 1999), 381–391. https://doi.org/10.1023/A:1010095604496

[15] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-Vectors and Arrays. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4590)*, Werner Damm and Holger Hermanns (Eds.). Springer, 519–531. https://doi.org/10.1007/978-3-540-73368-3_52

[16] Robert Glück and Jesper Jørgensen. 1995. Efficient multi-level generating extensions for program specialization. In *Programming Languages: Implementations, Logics and Programs*, Manuel Hermenegildo

and S. Doaitse Swierstra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 259–278.

[17] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[18] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS*. The Internet Society.

[19] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.* Prentice Hall.

[20] Ulrik Jørring and William L. Scherlis. 1986. Compilers and Staging Transformations. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida) *(POPL '86)*. Association for Computing Machinery, New York, NY, USA, 86–96. https://doi.org/10.1145/512644.512652

[21] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[22] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml - System Description. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8475)*, Michael Codish and Eijiro Sumii (Eds.). Springer, 86–102. https://doi.org/10.1007/978-3-319-07151-0_6

[23] S. C. Kleene. 1938. On Notation for Ordinal Numbers. *J. Symbolic Logic* 3, 4 (12 1938), 150–155. https://projecteuclid.org:443/euclid.jsl/1183385485

[24] Kin-Keung Ma, Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *SAS (Lecture Notes in Computer Science, Vol. 6887)*. Springer, 95–111.

[25] Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *ICSE*. IEEE Computer Society, 416–426.

[26] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 225–242. https://doi.org/10.1145/3341301.3359641

[27] Zachary Palmer, Theodore Park, Scott Smith, and Shiwei Weng. 2020. Higher-Order Demand-Driven Symbolic Evaluation. *Proc. ACM Program. Lang.* 4, ICFP, Article 102 (Aug. 2020), 28 pages. https://doi.org/10.1145/3408984

[28] Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. 2018. Unifying analytic and statically-typed quasiquotes. *Proc. ACM Program. Lang.* 2, POPL (2018), 13:1–13:33. https://doi.org/10.1145/3158101

[29] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[30] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *NDSS 2021, Network and Distributed System Security Symposium, 21-24 February 2021, Virtual Conference*, ISOC (Ed.).

[31] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*, Eelco Visser and Jaakko Järvi (Eds.). ACM, 127–136. https://doi.org/10.1145/1868294.1868314

[32] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal) *(ESEC/FSE-13)*. Association for Computing Machinery, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[33] Tim Sheard and Simon Peyton Jones. 2002. Template Metaprogramming for Haskell. *SIGPLAN Not.* 37, 12 (Dec. 2002), 60–75. https://doi.org/10.1145/636517.636528

[34] Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A practical unification of multi-stage programming and macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2018, Boston, MA, USA, November 5-6, 2018*, Eric Van Wyk and Tiark Rompf (Eds.). ACM, 14–27. https://doi.org/10.1145/3278122.3278139

[35] Walid Taha. 1999. *Multi-stage programming: Its theory and applications.* Ph. D. Dissertation. Oregon Graduate Institute of Science and Technology.

[36] Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97), Amsterdam, The Netherlands, June 12-13, 1997*, John P. Gallagher, Charles Consel, and A. Michael Berman (Eds.). ACM, 203–217. https://doi.org/10.1145/258993.259019

[37] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0

[38] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 530–541. https://doi.org/10.1145/2594291.2594340

[39] Guannan Wei, Oliver Bračevac, Shangyin Tan, and Tiark Rompf. 2020. Compiling Symbolic Execution with Staging and Algebraic Effects. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 164 (Nov. 2020), 33 pages. https://doi.org/10.1145/3428232

[40] Guannan Wei, Yuxuan Chen, and Tiark Rompf. 2019. Staged Abstract Interpreters: Fast and Modular Whole-program Analysis via Metaprogramming. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 126 (Oct. 2019), 32 pages. https://doi.org/10.1145/3360552

[41] Guannan Wei, Shangyin Tan, Oliver Bračevac, and Tiark Rompf. 2021. LLSC: A Parallel Symbolic Execution Compiler for LLVM IR. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1495–1499. https://doi.org/10.1145/3468264.3473108

[42] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

A Appendix

```
void f(int a, int b) {
  int sum = 0;
  for (int i = 0; i < 4; i++) {
    int n = b % 2;
    if (n) sum += 1;
    b /= 2;
  }
  if (a == 500 && sum == 4) target();
}

int main () {
  int b; make_symbolic(&b);
  for (int i = 0; i < 1000; i++) f(i, b);
}
```

Figure 5. p4-ComplexLoop, a difficult program for forward SE.