Bipath: Just-In-Time Incremental Symbolic Execution

David Williams-King and Riley Spahn
Columbia University

1 Introduction

Symbolic execution is a valuable method for achieving high test coverage in complex real world software. Systems such as Klee [2] consistently generate tests that cover a larger faction of code than developers writing tests, and are able to find bugs that are infeasible to catch using other methods. Unfortunately, symbolic execution is very slow to run on real world programs because it must explore the entire state space for any value that interacts with a symbolic variable. This leads to an exponential number of paths to be explored, so full symbolic exploration is often infeasible. Many or even most of these paths will not be executed during common usage and do not need to be validated ahead of time so it is possible to prune or use heuristics to guide symbolic execution systems. We propose Bipath, a new system built on Klee to perform incremental symbolic execution.

While most symbolic execution systems ask "Is there any way to make this program crash?", Bipath seeks to answer the question "Will this program crash with inputs like this one?" Rather than exploring all code paths to determine if there is any input that will cause a program to crash, Bipath determines if there are any serious bugs on paths that will be executed by the current input and ones similar to it. Bipath amortizes the large cost of symbolic execution across the lifetime of the program by exploring paths in a greedy manner without compromising symbolic execution's correctness guarantees.

Bipath's approach to software quality assurance is different than the traditional approach to QA. This difference is shown in Figure 1. Bipath is a form of "in the field" quality assurance. Rather than fully verifying or fully testing an application before it is deployed an application running under Bipath will continue to verify itself as it runs. Bipath is similar to but distinct from other in the field testing work such as In Vivo Testing [4]. While In Vivo Testing relies on the pre-existing unit tests that are written by developers executing in the field, Bipath relies on symbolic execution. In Vivo Testing's fault finding ability is limited to that of the application's unit tests while Bipath guarantees to explore states generated by the concrete input however, because the symbolic states are generated by symbolic inputs Bipath cannot guarantee to explore known edge cases ahead of time until the program is provided a similar input.

2 Approach

One key parameter for incremental symbolic execution is the granularity at which to split up symbolic execution. Since the execution state space is exponentially large, we must choose the size of each *execution unit* carefully. Units must be large enough that a single increment provides meaningful information and yet small enough that Klee can symbolically check a unit in a reasonable amount of time.

The simplest solution would be to symbolically execute precisely the concrete input given; but this will not yield any reusable information, and would require checking for every new case. A simple generalization policy would be to check at the fine-grained level of individual basic blocks or functions. As execution enters a basic block Bipath would halt execution, verify that particular function or basic block and then allow execution to continue. Bipath would verify the function or basic block by changing a subset of the defined state to be symbolic, whether that be function arguments or globally-accessible variables, and then passing the state to Klee for verification. We considered this approach but decided against it, because it would likely have been cost prohibitive. The system would need to constantly context switch between Klee and Bipath, determining if the current function call or basic block execution matches some known good constraints.

One possible approach to alleviate the overhead associated with fine-grained increments would be to only verify each basic block or function with some probability. This approach would provide weak guarantees on a single program invocation but over the long term would likely check most of the program. We decided against this approach because it undermines the otherwise strong guarantees possible when utilizing symbolic execution.

Bipath targets command-line applications such as `ls` or `du` that run relatively quickly. So we chose to use a coarse-grained approach that considers the entire program, deriving symbolic program inputs from the inputs and arguments provided by the user. Bipath uses Klee to symbolically check the the program with the constrained symbolic execution that will be some tight superset of the user-provided inputs. For longer running programs, we envision checkpoints throughout their execution at which symbolic execution could begin, or prioritizing symbolic

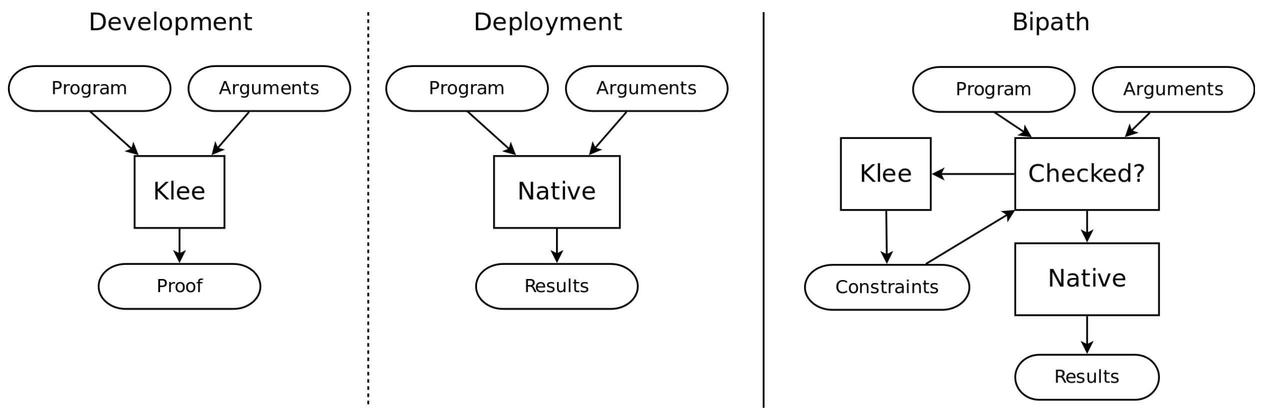

Figure 1: **Traditional vs Bipath QA Structure** Traditionally software development and quality assurance are conducted separately from deployment. Bipath combines execution and verification into one step.

inputs that are closest to what the user provides allowing the user's concrete execution to begin while symbolic execution continues in the background. This provides an easily-understandable partition of the input space, and hopefully leads to a reasonable partition of the execution state space.

The decision to use fine-grained or coarse-grained verification units is a design tradeoff. The overhead of constructing and checking constraints—and even the overhead of switching to Klee to perform the checking—can be very high, as we show later in the paper in Section 4. If the fraction of execution space that is checked is large it will minimize the cost to context switch, but it becomes more likely that the symbolic execution will not complete in a reasonable amount of time (if ever). The granularity we have chosen seems to be appropriate for the small command-line programs that we are targeting.

3 Architecture and Implementation

In order to build a system that incrementally symbolically checks programs, we need several components. As shown in Figure 2, we have separated the design into the following:

1. a *generalization* component, which abstracts particular concrete arguments into some symbolic test case to execute;

2. a *caching* or cache ordering component, which selects previous test cases to see if they are applicable; and

3. a *query transformation* component, which converts test cases output by Klee into the format necessary for rechecking them.

A Bipath program invocation will be executed as follows. First, Bipath will search the constraint cache and determine if the arguments passed to the program satisfy any of that program's known constraints. Bipath carries this out by using the query transformer to transform the known program constraints into concrete queries with the program arguments as input. For each of these queries Bipath invokes Klee's constraint solver, Kleaver, to determine if the input satisfies that constraint. If Bipath finds that the inputs match a known set of constraints (a cache hit), it will go on to natively execute the program. If Bipath does not find a matching constraint (a cache miss), then Bipath will generalize the arguments and invoke Klee to verify the program before native execution.

Argument Generalization

Currently, the Bipath generalization component simply takes concrete arguments and replaces them with symbolic arguments of the same length.

Constraint Cache

The constraint cache subsystem is charged with managing all known constraints for a program and determining if concrete program input satisfies one of the known constraints. This cache search algorithm is the key to Bipath's performance. We show in Section 4 that query evaluation throughput is a major bottle neck to Bipath performance so evaluating as few queries as possible is essential. The simplest cache search algorithm is to simply scan through all known constraints in a linear fashion. This is an inefficient algorithm and does not take advantage of Bipath's knowledge of constraint generalization. Another possible implementation is to use Bipath's knowledge of argument generalization and prioritize constraints known to be similar to the invocation ar-

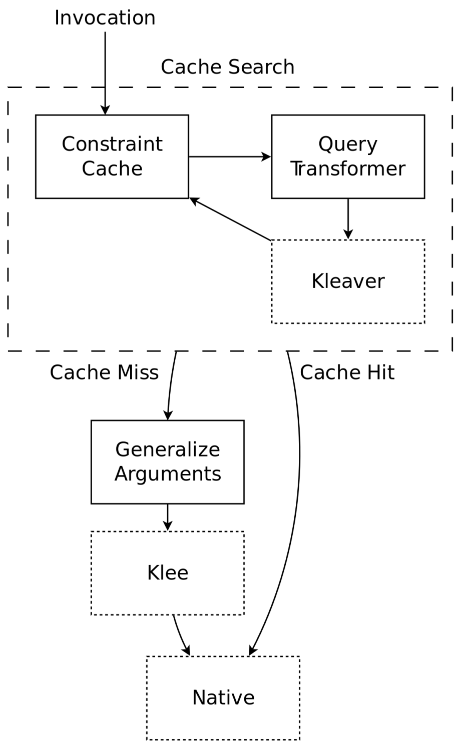

Figure 2: **Bipath Architecture** Boxes with solid outlines are components of Bipath.

guments. However, we did not want to restrict ourselves to a single generalization component and have not implemented the constraint search in this fashion. We chose to use a temporal cache that first searches the most recently matched constraints, following the intuition that users will often invoke a command with similar arguments.

Query Transformation

Bipath must transform constraints that Klee outputs into concrete queries about a specific input. Klee outputs all constraints found to hold as queries, each with a unique set of preconditions encoding that constraint. If a precondition does not in fact hold the constraint solver's behavior is undefined. An example of a Klee constraint without symbolic arguments follows in Figure 3.

```
(query [(Eq 1
  (ReadLSB w32 0 model_version))
  (Eq 45 (Read w8 0 arg0))
  ...
  (Extract w8 0 (SExt w32 N0)))]
false)
```

Figure 3: An example case. `(Eq 45 ...)` is testing if the first character of the argument is `'-'`.

But we are trying to check if any of the known constraints hold for a specific concrete input. We transform the preconditions into a single query where all preconditions from the Klee constraint are combined with `And` operators. The transformer changes the previously symbolic argument arrays (not shown below) into concrete values. The new query has no preconditions as we are trying to determine if a concrete input satisfies the constraint. An example Bipath query is shown here in Figure 4.

```
(query []
(And (And (And ...
  (Eq 1 (ReadLSB w32 0 model_version))
  (Eq 45 (Read w8 0 arg0))
  ...
  (Extract w8 0 (SExt w32 N0)))
```

Figure 4: The transformed Bipath query concatenating all preconditions into a single term.

Another possibility would be to leave the constraints as preconditions and the argument arrays symbolic, and introduce many new constraints that essentially define each character of the arrays; this may be less efficient as the solver may not be able to apply some optimizations reserved for concrete variables. Another possibility is to combine several such input cases into one much larger query, merging them together with `Or` operators. This would reduce the number of invocations of the constraint solver, at the cost of making each query more complex, and would reduce the effectiveness of cache search algorithms.

Usage Example

Bipath is implemented in two parts. The first is the Perl run harness that encompasses the constraint cache implementation and argument generalization. The query transformation is implemented partially as a C++ tool that uses Klee libraries for constraint manipulation, and partially within the Perl run harness. Bipath is executed as follows:

```
bipath-run.pl ./ls -lh
```

The first time the command is executed under Bipath, Bipath will invoke Klee with generalized arguments and write out the generated constraints for future use. For each successive invocation the run harness will invoke the C++ query transformation tool to search for constraints applicable to the current input. With the most recently used cache Bipath will test the last successful case immediately.

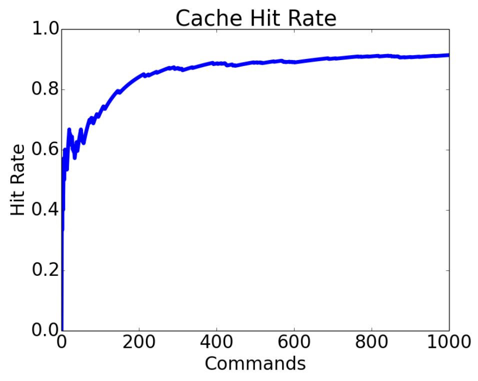

Figure 5: **Constraint Cache Hit Rate** The constraint hit rate is consistently near 0.9.

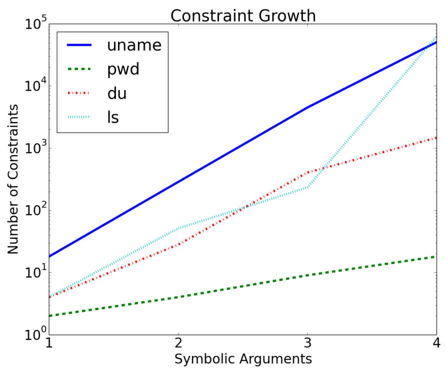

Figure 6: **Constraint Growth** The number of constraints generated grows exponentially with the number of symbolic arguments.

4 Evaluation

We evaluate Bipath in four ways. First, we evaluate a user's shell interactions to determine if users use utilities in such a way that Bipath will be beneficial. Second, we explore the number of constraints generated by a few common utilities to get an idea for amount of data we expect Bipath to need to process. Third, we look at query throughput for query transformation and query evaluation because we expect this to be the bottle neck for performance. Last, we examine Bipath's performance under different constraint search algorithms.

Shell Case Study

Our approach to Bipath assumes that users will execute applications and utilities many times with similar flags and inputs. If users do not use similar flags and inputs then Bipath will not have cached the required constraints and will often be required to execute Klee and symbolically verify inputs. To validate this assumption we performed a small case study on a single user's shell history. The observed shell history consisted of approximately one thousand commands that executed 32 unique utilities found in either the GNU Core Utilities [3] or Busy Box [1].

We evaluated the constraint cache hit rate by examining each command and determining if the constraints required to check the command would have been cached by Bipath. The results of this study can be found in Figure 5. The x axis in the figure represents the number of commands executed and the y axis represents the fraction of commands that would have been cached.

The results of the experiment show that a large proportion of commands, greater than 0.9, will be cached by Bipath under this usage model after approximately 700 commands. We could potentially warm the cache by executing a small test suite for common utilities before production use.

Constraint Growth

Bipath performance is heavily influenced by the number of constraints that need to be evaluated before native execution (in essence, the number of wrong guesses by the cache ordering component). To evaluate the potential number of required constraints we observed the number of constraints Klee generates for four utilities: uname, pwd, du, and ls, with symbolic arguments of lengths one through four. Figure 6 shows the results of this study. The x axis represents the symbolic argument length and the y axis represents the number of constraints generated symbolic arguments of that length in log base ten scale.

The number of generated constraints grows exponentially with the length of the symbolic arguments. With symbolic arguments of length four, uname and ls both generate more than 50 thousand constraints that may need to be checked.

Query Throughput

Bipath's performance is limited by how fast it can find a known constraint that matches the current input, as this time will dwarf the time required for executing the actual command. We evaluated the throughput of Bipath's query transformer and Kleaver, the Klee tool we use for query evaluation. The results of this evaluation are shown in Table 1. We observed that the Bipath query transformer achieves 782 queries per second and Kleaver evaluates 145 queries per second.

To provide context for these throughputs we examine the command `ls -lah`. Figure 6 shows that the `ls` command with three symbolic arguments generates

Subsystem	**Queries / Second**
Query Transformer	782
Query Evaluation (Kleaver)	145

Table 1: **Bipath Throughput** The throughput of Bipath subsystems in queries per second.

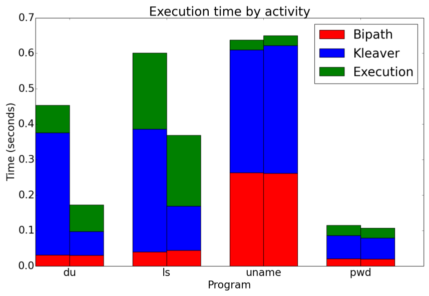

Figure 7: **Linear and MRU Search Performance** The figure shows linear search performance on the left and MRU search performance on the right for each of the utilities.

approximately four thousand constraints. When executing `ls -lah` under Bipath native execution does not proceed until a matching constraint has been found. If Bipath searches randomly we expect that it will need to test two thousand queries before finding a matching constraint. Assuming Bipath needs to transform and evaluate two thousand queries at 782 and 145 queries per second respectively, we expect that it will be approximately 16.4 seconds before `ls -lah` begins native execution. If Bipath caches the transformed queries we can still expect it to need to spend 13.8 seconds evaluating queries before proceeding to native execution. Fortunately we can lower this significantly in real world use by tuning the constraint search algorithm. For example, if the most-recently-used ordering heuristic is applied, the very first constraint may succeed, resulting in only eight milliseconds of overhead.

Bipath Performance

We evaluated Bipath's performance with both the linear search strategy and the most recently used (MRU) algorithm. The linear search strategy is implemented by simply scanning through and testing the constraints in the order that they were produced by Klee. The MRU search maintains a list of the most recently matched constraints for a given command or utility. When a constraint is matched it is moved to the head of the list and removed from the tail of the list if it is present. Our current implementation does not cache transformed queries so we do not expect to see any speed up from reduction in query transformations.

To evaluate both of the search algorithms we generated workloads used to perform the case study in Section 4. We observed performance increases in three of the four utilities as shown in Figure 7. Figure 7 shows the amount of time four different utilities spend in each phase of Bipath execution under linear search, the left bar, and MRU search, the right bar. Overall performance gains ranged from 61% improvement in `du` to 7% overall improvement in `pwd` with an average improvement of 26% over linear search. `du` saw the greatest improvement reducing it's time spent executing query evaluations by 80% from 0.35 seconds to 0.068 seconds. Under real world usage patterns the most recently used search provides Bipath a significant improvement over the linear search implementation.

5 Future Work

In this paper we have focused exclusively on the technique of symbolic execution. However, the idea of partitioning the work necessary for a static analysis and deciding at runtime which parts of the space to explore could have broader application. Techniques such as array bounds checking could be disabled once a function is known to not buffer overrun on some inputs. These ideas have be applied for performance optimizations for many years, by just-in-time compilers that decide where to focus their work based on runtime characteristics. We believe that the same is possible for many static analysis and security properties, and in the future look forward to exploring these directions.

References

[1] Busy Box. `http://www.busybox.net/`.

[2] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[3] GNU Coreutils. `https://www.gnu.org/software/coreutils/`.

[4] Christian Murphy, Gail Kaiser, Ian Vo, and Matt Chu. Quality assurance of software applications using the in vivo testing approach. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, 2009.