Prototyping Symbolic Execution Engines for Interpreted Languages

Stefan Bucur
École Polytechnique Fédérale de Lausanne
stefan.bucur@epfl.ch

Johannes Kinder
École Polytechnique Fédérale de Lausanne
Royal Holloway, University of London
johannes.kinder@rhul.ac.uk

George Candea
École Polytechnique Fédérale de Lausanne
george.candea@epfl.ch

Abstract

Symbolic execution is being successfully used to automatically test statically compiled code [4, 7, 9, 15]. However, increasingly more systems and applications are written in dynamic interpreted languages like Python. Building a new symbolic execution engine is a monumental effort, and so is keeping it up-to-date as the target language evolves. Furthermore, ambiguous language specifications lead to their implementation in a symbolic execution engine potentially differing from the production interpreter in subtle ways.

We address these challenges by flipping the problem and using the interpreter itself as a specification of the language semantics. We present a recipe and tool (called CHEF) for turning a vanilla interpreter into a sound and complete symbolic execution engine. CHEF symbolically executes the target program by symbolically executing the interpreter's binary while exploiting inferred knowledge about the program's high-level structure.

Using CHEF, we developed a symbolic execution engine for Python in 5 person-days and one for Lua in 3 person-days. They offer complete and faithful coverage of language features in a way that keeps up with future language versions at near-zero cost. CHEF-produced engines are up to 1000× more performant than if directly executing the interpreter symbolically without CHEF.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—symbolic execution; D.3.4 [*Programming Languages*]: Processors—interpreters

Keywords state selection strategies; software analysis optimizations; interpreter instrumentation

ASPLOS '14, March 1–4, 2014, Salt Lake City, Utah, USA.

ACM 978-1-4503-2305-5/14/03. . . $15.00.
http://dx.doi.org/10.1145/2541940.2541977

Reprinted from ASPLOS '14, Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, March 1–4, 2014, Salt Lake City, Utah, USA

1. Introduction

Developers spend much of their time writing unit and performance tests and manually tracing hard-to-reproduce bugs; ideally, these tasks would be automated. Symbolic execution is a particularly successful technique for exploring multiple execution paths fully automatically. It has been used to find bugs and to generate high-coverage test suites for Windows media format parsers [16], Windows device drivers [19], and the Linux Coreutils [7], and it has helped to reproduce crashes for debugging [29]. Symbolic execution enumerates feasible execution paths by using a constraint solver to synthesize inputs that drive the program down paths that have not been covered before.

Most mature systems for symbolic execution target low-level code representations. The widely used symbolic execution engines KLEE [7], SAGE [16], Bitblaze [25], and S2E [9] all process either x86 machine code or LLVM [21] intermediate representation code. This makes them applicable to languages that are statically compiled to one of these representations.

However, an increasingly important class of software is never statically compiled: code in languages like Python, Perl, Lua, or Bash is directly executed by interpreters, and its dynamic features generally prevent static compilation. These languages are popular because they enable rapid prototyping and are easy to use; as a result, they are increasingly used in infrastructure and systems code [3, 18].

Consequently, there have been initial efforts to directly implement high-level symbolic execution engines for interpreted languages. For example, the NICE engine for Python [8] symbolically executes OpenFlow controller code, and the Kudzu engine for JavaScript [24] finds code injection vulnerabilities in web applications.

The implementation of a dedicated symbolic execution engine is a significant undertaking for each language. A symbolic execution engine is essentially a fully fledged interpreter for the language, operating on constraints and expressions instead of concrete memory objects. This is why existing dedicated engines usually do not support general programs in the full target language. They either support

only a language subset that is relevant for a particular application domain [8, 24], or they require the developer to write their code to explicitly use custom types and libraries that provide symbolic execution functionality [11].

Even supporting just a portion of the language typically involves reading language specifications and carefully writing the engine to faithfully implement them. Unfortunately, language specifications are often imprecise or deliberately leave implementation choices to the developers of the interpreter, which gives rise to multiple dialects. Quoting the Python Language Reference: *"Consequently, if you were coming from Mars and tried to re-implement Python from this document alone, you might have to guess things and in fact you would probably end up implementing quite a different language."* [22, §1] Combined with the fact that languages like Ruby and Python are continuously evolving, this makes writing and maintaining language-specific symbolic execution engines a daunting task, thus excluding many languages from the benefits of symbolic execution-based analysis and automated testing.

In this paper, we propose and demonstrate the idea of building a symbolic execution engine for an interpreted language *from its interpreter*. The interpreter is the place where all subtleties of the language are precisely encoded, and it is the only specification that matters in practice.

We implement this idea in CHEF, a tool that takes a specially-packaged interpreter as input and becomes a fully functional symbolic execution engine for the interpreter's target language. CHEF's infrastructure for symbolic execution is language-agnostic and tailored for interpreters with a moderately-sized core (10s of KLOC), in the ballpark of Python, Ruby, and Lua. Languages as complex and large as Java are not yet a good target for CHEF. The interpreter has to be instrumented using a lightweight API to interact with CHEF.

Using CHEF, we developed symbolic execution engines for Python and Lua. In comparison to the cost of implementing a faithful new engine from scratch, adapting the language interpreters was orders of magnitude easier: for Python, 321 lines of code had to be added to the interpreter and took 5 person-days, while Lua required 277 lines of code written in 3 person-days.

We evaluated the two engines on 11 popular Python and Lua library packages, where they generated up to 1000× more test cases compared to applying plain symbolic execution on the interpreter executable. The generated tests obtained good coverage results and uncovered a number of bugs. We also compared the Python engine to dedicated implementations and found that its slower execution speed is balanced by more complete and accurate support of language features.

This paper makes two contributions:

- We show how to derive high-level symbolic execution of an interpreted language from low-level symbolic execution of the language's interpreter, while preserving full soundness and completeness.In addition, low-level symbolic execution allows to seamlessly support native library code.
- We introduce class-uniform path analysis (CUPA), a state selection heuristic that is crucial for a meaningful exploration of the interpreted program. CUPA partitions the set of symbolic execution states into classes based on their corresponding high-level program path. By adjusting the state selection probability of each class, CUPA ensures steady progress in the exploration of the interpreted program.

The remainder of the paper is organized as follows: we give background on symbolic execution and challenges related to interpreted languages (§2). We then describe CHEF and the CUPA heuristic (§3), followed by a recipe for preparing an interpreter for use with CHEF (§4). We describe how the symbolic execution engines for Python and Lua were produced (§5) and evaluate them (§6). Finally, we discuss related work (§7) and conclude (§8).

2. Symbolic Execution of Interpreted Code

We briefly describe symbolic execution and its use for test generation (§2.1). We then explain the problems in symbolically executing interpreted languages (§2.2). Finally, we present our idea of executing the target program through its interpreter and the challenges that arise from this (§2.3).

2.1 Symbolic Execution

The intuition behind symbolic execution is to execute target code with *symbolic* instead of concrete input values, to cover the entire input space and thus all possible paths. For instance, a function `foo(int x)` is executed with a symbolic variable λ assigned to `x`. Statements using `x` manipulate it symbolically: `x := x * 3` updates the value of `x` to $3 \cdot \lambda$. The mappings from variables to expressions are kept in a *symbolic store* during symbolic execution.

Path Condition Besides the symbolic store, the symbolic program state maintains a *path condition*. Whenever symbolic execution takes a conditional branch, the condition (or its negation, for `else` branches) is added to the path condition. Continuing our example, execution along the `true` branch of `if (x > 10)` would record $3 \cdot \lambda > 10$ in the path condition. Thus, each symbolic execution state represents a particular path through the program, and its path condition encodes the input constraints for taking this path.

Dynamic Test Generation Dynamic test generation in the spirit of DART [15] (often referred to as *concolic* execution) executes the program symbolically along a single path that is given by a concrete input assignment. This has the benefit of reaching deeper program behaviors and reducing the number of constraint solver invocations. To this end, the symbolic execution state also contains a concrete store that keeps a

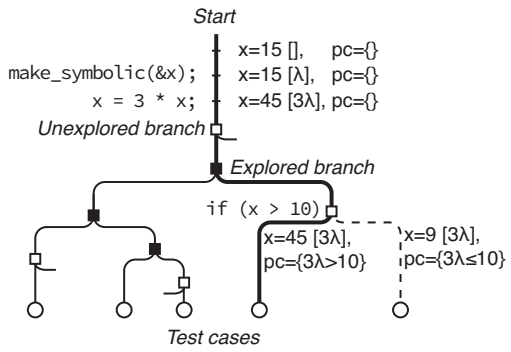

Figure 1. Symbolic execution tree for a simple example. States maintain concrete and symbolic values for `x` and carry a path condition (pc). The bold line represents the current concolic path, and boxes are branching points. When a candidate alternate branch (empty box) is selected, new input assignments are computed that satisfy the path condition of the new path (dashed line).

regular concrete value for each variable. We use a slight variation that avoids re-executing common path prefixes.

Consider the tree representation of program paths for the example in Figure 1. The current concrete path is the bold line. At each conditional branch, the symbolic execution engine forks the symbolic execution state. If the `then` branch is implied by the concrete value in the current state, the negated branch condition (for the `else` branch) is added to the path condition of the alternate state, and vice versa. The alternate state is tentatively added to the symbolic execution tree (empty boxes in Figure 1) for later exploration.

Once the concrete path reaches its end (the circle), the symbolic execution engine executes a new path (the dotted line) by picking an alternate state and asking a constraint solver whether the path condition of the alternate state is satisfiable. If it is, the satisfying assignment of input variables constitutes a test case that would take the program down this new path. Plugging the assignment into the expressions of the symbolic store yields a new concrete store for the alternate path. If the path condition is unsatisfiable, the path can never be taken at runtime—it is infeasible—and the corresponding state is discarded by the engine.

Search Strategies The main scalability challenge for symbolic execution is *path explosion*: since each conditional branch can potentially fork the execution, the number of states (and thus paths) grows roughly exponentially in the size of the program. Several approaches exist for reducing path explosion to some extent [14, 20] but, in general, it cannot be completely avoided.

Due to path explosion, exhaustive exploration is unrealistic for most systems code, so symbolic execution can only cover a subset of the possible paths. Modern symbolic execution engines therefore use *search strategies* to reach exploration goals (e.g., line coverage) quickly [6, 7, 16, 28]. The search strategy prioritizes alternate paths it deems "interesting". Thus, the usage model for automated test generation by symbolic execution is to run for a fixed amount of time and generate as many test cases as possible, or run until a fixed coverage goal is reached (e.g., 80% line coverage).

Like all heuristics, search strategies are not universal, and their effectiveness depends on the code being explored. One of the challenges of effectively generating test cases by symbolic execution is thus to find the strategy that leads to the testing goal as fast as possible on the target code.

2.2 Supporting Interpreted Languages

Building a correct and complete symbolic execution engine for an interpreted language is generally harder than building one for a low-level language. Statements of interpreted languages can wrap complex operations that, in lower-level languages, would be implemented through libraries. For instance, Python strings are a built-in type offering more than 30 operations (such as `find`) as part of the language, implemented natively in the interpreter. Other language features that allow to inspect or even modify the code itself, i.e., runtime reflection, are even more tedious to implement and very hard to get right.

Finally, besides requiring an enormous initial effort to build a symbolic execution engine that fully supports them, dynamic languages also evolve fast. This implies constant, labor-intensive maintenance and co-evolution of the symbolic execution engine, if it is to keep up with the newest versions of the language.

2.3 Symbolically Executing the Interpreter

Considering the difficulty of directly supporting interpreted languages, we resort to symbolically executing the interpreter itself, since it completely defines the semantics of the target language as a function of the semantics of the language the interpreter is implemented in. After compiling the interpreter to a format supported by an existing symbolic execution engine, one can symbolically execute an interpreted program by symbolically executing the interpreter with the target program as argument. However, even though in principle this direct approach yields a symbolic execution engine for the target language, it is impractical, due to the engine not being aware of the control flow of the interpreted program.

High- vs. Low-level Program Paths An interpreted program conceptually executes both on a high level—the level of the target language—and a low level—the level of the interpreter. A high-level program path is a sequence of values of the high-level program counter (HLPC). Each HLPC value corresponds to a program statement or bytecode instruction (both Python and Lua use intermediate bytecode). Branches can occur explicitly at control flow statements, or implicitly through exceptions. A low-level program path is a sequence of machine instructions from the interpreter binary, includ-

```
1 def validateEmail(email):
2   at_sign_pos = email.find("@")
3   ...
4   if at_sign_pos < 3:
5     raise InvalidEmailError()
6   ...
```

```
1 def average(x, y):
2   return (x + y) / 2
```

Figure 2. Two examples of Python code that lead to path explosion when the interpreter running it is symbolically executed.

ing its code for internal bookkeeping (e.g., details of reference counting and garbage collection).

Due to the additional implementation details, a single high-level path can map to multiple low-level paths. Figure 2 shows two examples of Python code that have few high-level but many low-level paths. The **`validateEmail`** method has only two high-level paths, but its use of **`string.find`** leads to as many low-level paths as there can be characters in the **`email`** string. The second example **`average`** may come as more of a surprise: even though it has just a single high-level path, symbolic execution can end up enumerating many low-level paths: Python uses arbitrary-precision integers, so the interpreter may have to iterate over digit vectors of arbitrary length, which can in principle spawn arbitrarily many paths.

Challenges for Search Strategies The search strategy of a low-level symbolic execution engine is oblivious to the high-level program structure of the target program, and it essentially just tries to cover the interpreter. This generally leads to covering the same high-level paths many times with multiple distinct low-level paths. For instance, a high-level statement like **`find`** can lead to hundreds of alternate states, whereas a primitive integer comparison might just create a single one. Therefore, the low-level search strategy is likely to explore multiple ways for **`find`** to succeed or fail, without increasing high-level coverage, before eventually exploring the alternate outcome of the comparison.

The key is to make the engine aware of the high-level interpreted program. By tracing the values of the HLPC, the engine can construct a high-level control flow graph (CFG) on the fly that can be be leveraged by the search strategy.

Alas, a strategy cannot straightforwardly determine future branching points in a high-level CFG: two low-level paths can fork from the same prefix *before* their corresponding high-level paths do. This can be due to having distinct bytecode instructions for comparisons and conditional jumps, or due to native library calls. In Figure 3, three low-level paths fork within the single HLPC location for **`email.find`**. The low-level paths remain on the same high-level path until reaching the branching HLPC, where they diverge into two distinct high-level paths. The relevant alternate low-level states for covering the distinct high-level paths thus were located away from the location of the code

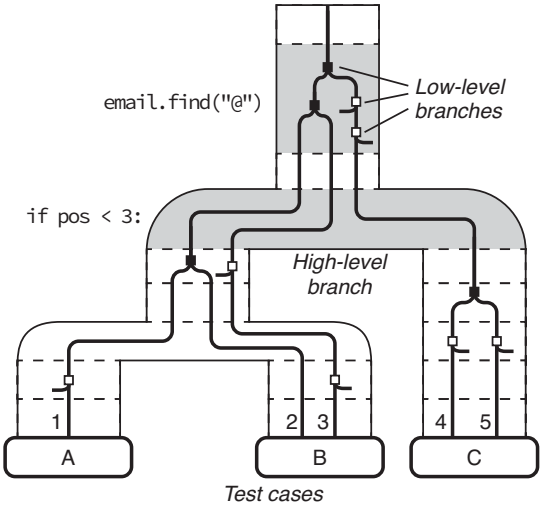

Figure 3. High-level execution tree (paths A, B, and C), as induced by its low-level execution paths (1–5) for the first running example in Figure 2. Dotted lines segment high-level execution paths into bytecode instructions. One high-level path may correspond to multiple low-level paths explored.

interpreting the high-level control flow statement. The issue of pre-determining branches is present also when exploring regular code, but it is ubiquitous when exploring code on interpreters.

3. The CHEF System

We now present the architecture of CHEF (§3.1) and introduce CUPA, our state selection mechanism (§3.2). We then describe CUPA optimized for exploring distinct high-level paths (§3.3) and optimized for high line coverage (§3.4).

3.1 System Overview

CHEF is a platform for language-specific symbolic execution. Provided with an interpreter environment, which acts as an executable language specification, it becomes a symbolic execution engine for the target language (see Figure 4). The resulting engine can be used like a hand-written one, in particular for test case generation. When fed with a target program and a symbolic test case (also called test driver or test specification in the literature), it outputs a set of concrete test cases, as shown in Figure 4.

CHEF is built on top of the S2E analysis platform [9]. S2E symbolically executes a virtual machine containing the interpreter and a testing library at the level of machine code, including the OS kernel, drivers, and user programs. S2E provides an API that guest code can use to declare memory buffers as symbolic. Comparisons on symbolic values cause S2E to fork new paths, which are enqueued and explored following a search strategy. CHEF extends the S2E guest API with a high-level instruction instrumentation call (§4.1), in-

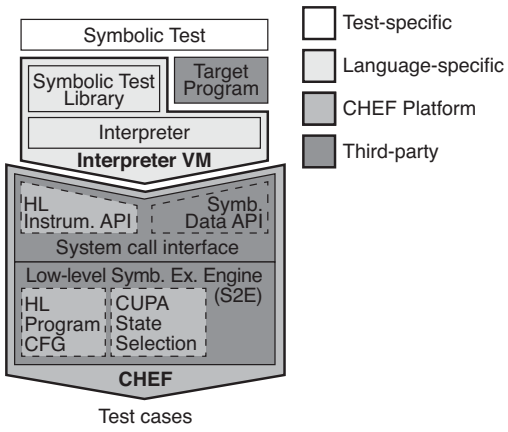

Figure 4. Schema of CHEF's architecture.

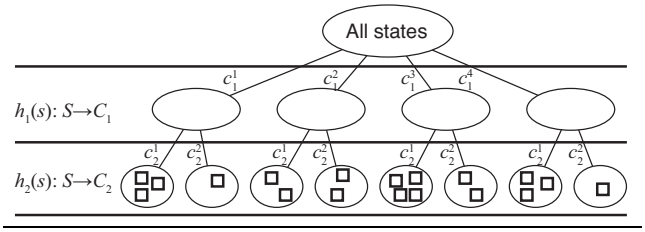

Figure 5. CUPA state partitioning. Each level corresponds to a state classification scheme. Child nodes partition the parent node according to the classification at their level.

voked by interpreters to trace the currently executing high-level path. The explored high-level paths are used to construct a high-level execution tree and a low-level to high-level mapping (i.e., the data structure shown in Figure 3). CHEF uses a state selection strategy to maximize the ratio of high-level to low-level paths (§3.2).

The resulting engine is a correct symbolic execution engine for the target language *as defined by the interpreter*. It is fully precise and theoretically complete, i.e., it will not explore infeasible paths and will eventually explore all paths. The usual limitations of symbolic execution engines apply: completeness holds only under the assumption that the constraint solver can reason about all generated path conditions, and it is understood that exhaustive exploration is usually impossible in finite time.

3.2 Class-Uniform Path Analysis (CUPA)

Consider using symbolic execution for achieving statement coverage on a program containing a function with an input-dependent loop. At each iteration, the loop forks one additional state (or exponentially many, if there are branches in the loop). A strategy that selects states to explore uniformly is therefore biased toward selecting more states from this function, at the expense of states in other functions that fork less but contribute equally to the statement coverage goal.

We reduce this bias by introducing Class-Uniform Path Analysis (CUPA). The main idea is to group states into classes and then choose uniformly among classes instead of states. For instance, in the above example, the class of each state could be its current function. CUPA then first selects uniformly a function, then picks at random a state inside that function. This way, functions generating many states are still selected with equal probability to others.

In general, CUPA organizes the state queue into a hierarchy of state subsets rooted at the entire state queue (see Figure 5). The children of each subset partition the subset according to the *state classification scheme* at their level. A classification scheme is defined as a function $h : S \to C$, where $h(s)$ maps each state s into a class value c. States of the same parent with the same class value are sorted into the same child. CUPA selects a new state for exploration by performing a random descent in the classification tree, starting from the root. When reaching a leaf, the strategy takes out a random state from the state set and returns it to the symbolic execution engine for exploration. By default, all sibling classes on each level have equal probability of being picked, but they can be assigned weights if required.

A CUPA strategy is parameterized by the number N of levels in the tree and a classification function h_i for each level $i = 1 \ldots N$. CHEF uses two instantiations of CUPA: one optimized for covering high-level paths (§3.3) and one for covering the high-level CFG, i.e., statements (§3.4).

3.3 Path-Optimized CUPA

A low-level strategy unaware of the high-level program would be implicitly biased towards picking high-level instructions that fork more low-level states than others, such as string operations or native calls. To mitigate this, we instantiate a two-level CUPA strategy using the following classes:

1. The location of the state in the high-level symbolic execution tree. This is the occurrence of the state's high-level program counter (HLPC) in the unfolded high-level CFG, referred to as the dynamic HLPC. We choose the dynamic HLPC to give each high-level path reaching the HLPC the same chance to fork and subsequently diverge.
2. The low-level x86 program counter of the state. This classification reduces the selection bias of "hot spots" of path explosion within a single complex instruction, such as a native function call.

3.4 Coverage-Optimized CUPA

Based on a coverage-optimized strategy introduced by the KLEE symbolic execution engine [7], we developed a CUPA instance that partitions states according to their minimum distance to branches leading to uncovered code. Alas, dynamic language interpreters do not generally have a static CFG view of the program, so code that has not been covered yet is not accessible to the search strategy. The high-level CFG of the target program is dynamically discovered along each execution path. On this CFG, we employ heuris-

tics that (1) identify the instruction opcodes that may branch, and (2) weigh the state selection toward states that are closer to these potential branching points.

First, CHEF identifies the branching opcodes by collecting all high-level instructions that terminate a basic block with an out-degree in the CFG of at least 2 (i.e., cause branching in the control flow). We then eliminate the 10% least frequent opcodes, which correspond to exceptions or other rare control-flow events. Second, CHEF identifies the potential branching points as those instructions in the CFG that have a branching opcode (as previously identified) but currently only one successor. Finally, CHEF computes for each execution state the distance in the CFG to the closest such potential branching point.

Having computed this information, we instantiate a two-level CUPA strategy with the following classes:

1. The static HLPC of the state in the high-level CFG. On this level, each class is weighted by $\frac{1}{d}$, where d is the distance in the inferred high-level CFG to the closest potential branching point, making states at locations close to a target more likely to be selected.
2. The state itself (so each partition has a single element). On this level, the states are weighted by their *fork weight*.

Fork weight is computed by counting the number of consecutive forks at the same low-level program counter (i.e., at an input-dependent loop in machine code). States $1,\ldots,n$ forking from the same path at the same location get weights $p^n, p^{n-1}, \ldots, 1$, where $p < 1$ de-emphasizes states forked earlier ($p = 0.75$ in our implementation). The last state to fork at a certain location thus gets maximum weight, because alternating the last decision in a loop is often the quickest way to reach different program behavior (e.g., to satisfy a string equality check).

4. Preparing the Interpreter

We now explain how to prepare an interpreter for CHEF: the first and mandatory step is to instrument the main interpreter loop to report HLPCs (§4.1); the second and optional step is to optimize the interpreter for efficient symbolic execution (§4.2). CHEF provides an API (Table 1) that will be explained along with its use. Finally, we discuss the remaining work of building a language-specific testing API to the resulting engine (§4.3).

4.1 Exposing the High-Level Program Location

To reconstruct the high-level program paths and CFG, CHEF needs to identify the high-level instructions executed on each low-level path. CHEF provides the `log_pc(pc, opcode)` API call to the interpreter, which declares the current high-level program location and the type (opcode) of the next instruction. A high-level instruction is executed in between two consecutive `log_pc` calls. Interpreters typically contain a main interpretation loop that `switch`-es over the type of the current instruction and invokes specific handlers. The `log_pc` call can be added conveniently at the head of the interpreter loop.

API Call	Description
`log_pc(pc, opcode)`	Log the interpreter PC and opcode
`start_symbolic()`	Start the symbolic execution
`end_symbolic()`	Terminate the symbolic state
`make_symbolic(buf)`	Make buffer symbolic
`concretize(buf)`	Concretize buffer of bytes
`upper_bound(value)`	Get maximum value for expression on current path
`is_symbolic(buf)`	Check if buffer is symbolic
`assume(expr)`	Assume constraint

Table 1. The CHEF API used by the interpreters running inside the S2E VM.

In our design, we make minimal assumptions about the language structure, so the HLPC and opcode values are opaque; the CUPA strategies were designed accordingly. Nonetheless, more specific versions of the system could add structure to the two values, e.g. provide a pair of function name and offset as HLPC. The additional information can be used by CHEF to improve the exploration heuristics (e.g., by creating a CUPA class).

The granularity of `log_pc` calls depends on the language structure. CHEF's correctness does not depend on the specific instrumentation pattern, but the more closely the reported HLPC corresponds to statements in the target program, the more accurately CUPA can cluster states. In the extreme, if `log_pc` is never invoked, CHEF would see the entire program as a single high-level instruction and lose the advantage of CUPA clustering for HLPCs.

4.2 Optimizing for Symbolic Execution

In order to maximize performance, interpreters make heavy use of special cases and sophisticated data structures. Unfortunately, these features hurt the performance of symbolic execution by amplifying path explosion and increasing the complexity of symbolic formulas [27].

We identify a number of easy optimizations that preserve the interpretation semantics but significantly improve symbolic execution performance. The optimizations use the CHEF API in the last block of rows in Table 1.

Neutralizing Hash Functions Hash functions are especially common in interpreters, due to the internal use of hash tables for associative data structures (e.g., Python dictionaries or Lua tables). However, they are generally a problem in symbolic execution: a symbolic value added to a hash table (a) creates constraints that essentially ask the constraint solver to reverse a hash function, which is often hard, and (b) causes the exploration to fork on each possible hash bucket the value could fall into. A simple and effective optimization is to *neutralize the hash function*, i.e., replace it with a degenerate one returning a single constant. This

```
void *malloc(size_t size) {
  if (is_symbolic(&size, sizeof(size))) {
    size_t upper_size = upper_bound(size);
    return old_malloc(upper_size);
  }
  return old_malloc(size);
}
```

Figure 6. Example of a symbolic execution-aware `malloc` function wrapper created using the CHEF API. If the allocation size is symbolic, the wrapper determines its upper bound and issues a concrete request to the underlying implementation.

change honors the usual contracts for hash functions (equal objects have equal hashes) and will turn hash lookups into list traversals.

Avoiding Symbolic Pointers Input-dependent pointers (also referred to as symbolic pointers) may point to multiple locations in the program memory, so a pointer dereference operation would have to be resolved for each possible location. In practice, symbolic execution engines deal with this situation in one of two ways: (a) fork the execution state for each possible concrete value the symbolic pointer can take; or (b) represent the dereference symbolically as a read operation from memory at a symbolic offset and let the constraint solver "deal" with it. Both ways hurt symbolic execution, either by causing excessive path explosion or by burdening the constraint solver.

While there is no generic way to avoid symbolic pointers other than concretizing their values (the `concretize` API call) at the price of losing completeness, there are specific cases where they can be avoided.

First, *the size of a buffer can be concretized* before allocation. A symbolic size would most likely cause a symbolic pointer to be returned, since a memory allocator computes the location of a new block based on the requested size. To avoid losing completeness, a symbolic execution-aware memory allocator can determine a (concrete) upper bound on the requested size and use that value for reserving space, while leaving the original size variable symbolic. This way, memory accesses to the allocated block would not risk being out of bounds. Figure 6 shows how the CHEF API is used to wrap a call to the `malloc` function in the standard C library.

Second, *caching and "interning" can be eliminated*. Caching computed results and value interning (i.e., ensuring that a single copy of each possible value of a type is created) are common ways to improve the performance of interpreters. Alas, when a particular value is computed, its location in memory becomes dependent on its value. If the value was already in the cache or in the interned store, it is returned from there, otherwise a new value is computed. During symbolic execution, this logic becomes embedded in the value of the returned pointer, which becomes symbolic. Disabling caching and interning may hurt the native performance of the program, but it can give a significant boost when running inside a symbolic execution engine.

Component	Python	Lua
Interpreter core size (C LoC)	427,435	14,553
HLPC instrumentation (C LoC)	47 (0.01%)	44 (0.30%)
Sym. optimizations (C LoC)	274 (0.06%)	233 (1.58%)
Native extensions (C LoC)	1,320 (0.31%)	154 (1.06%)
Test library (Python/Lua LoC)	103	87
Developer time (person-days)	5	3

Table 2. Summary of the effort required to support Python and Lua in CHEF. The first row is the interpreter size without the standard language library. The next two rows are changes in the interpreter core, while the following two constitute the symbolic test library. The last item indicates total developer effort.

Avoiding Fast Paths A common way to speed-up the native performance of a function is to handle different classes of inputs using faster specialized implementations ("fast paths"). For example, a string comparison automatically returns false if the two strings have different lengths, without resorting to byte-wise comparison.

Fast paths may hurt symbolic execution because they cause symbolic branches in the code checking for the special input conditions. *Eliminating short-circuited returns* can reduce path explosion. Instead of returning to the caller as soon as it produced an answer, the function continues running and stops on an input-independent condition. For example, when comparing two strings of concrete length, a byte-wise string comparison would then traverse the entire string buffers in a single execution path, instead of returning after the first difference found.

4.3 Testing API

Programs to be tested can be fed symbolic inputs by marking input buffers with `make_symbolic` and defining conditions over the input with the `assume` call, in accordance to the test specification. Note that the buffer is a memory region of concrete bounds. It is the job of the symbolic test library in the interpreter VM to convert from the language data structures (e.g., strings, integers) to the memory locations used to store the data in the interpreter implementation.

5. Case Studies

We used CHEF to generate symbolic execution engines for Python (§5.1) and Lua (§5.2). Table 2 summarizes the effort to set up the two interpreters for CHEF. The necessary changes to the interpreter amount to 321 lines of code for Python and 277 for Lua. The total developer time was 5 person-days for Python and 3 person-days for Lua, which is orders of magnitude smaller than the effort required for building a complete symbolic execution engine from scratch.

```
class ArgparseTest(SymbolicTest):
    def setUp(self):
        self.argparse = importlib.import_module("argparse")

    def runTest(self):
        parser = self.argparse.ArgumentParser()
        parser.add_argument(
            self.getString("arg1_name", '\x00'*3))
        parser.add_argument(
            self.getString("arg2_name", '\x00'*3))

        args = parser.parse_args([
           self.getString("arg1", '\x00'*3),
           self.getString("arg2", '\x00'*3)])
```

Figure 7. The symbolic test used to exercise the functionality of the Python `argparse` package.

5.1 Symbolic Execution Engine for Python

Interpreter Instrumentation We instrumented the CPython interpreter 2.7.3 for use with CHEF, according to the guidelines presented in §4.

Python programs are composed of modules, corresponding to Python source files. Each source file is compiled into an interpreter-specific bytecode format, i.e., each source statement is translated into one or more lower-level primitive instructions. The instructions are grouped into blocks, corresponding to a single loop nesting, function, method, class, or global module definition. We define an HLPC as the concatenation of the unique block address of the top frame on the stack and the current instruction offset inside the block. We instrumented the Python interpreter to pass this program location to CHEF; this required adding less than 50 LoC to the main interpreter loop.

We performed several optimizations on the Python interpreter: we neutralized the hash functions of strings and integers, which are the most common objects; we concretized the memory sizes passed to the garbage-collected memory allocator; and we eliminated interning for small integers and strings. Most optimizations involved only adding preprocessor directives for conditional compilation of blocks of code. We gathered the optimizations under a new `-with-symbex` flag of the interpreter's `./configure` script.

Symbolic Tests To validate the usefulness of the resulting symbolic execution engine, we use it as a test case generation tool. To this end, we implemented a symbolic test library as a separate Python package, used both inside the guest virtual machine, and outside, during test replay. Figure 7 is an example of a symbolic test class for the `argparse` command-line interface generator. It sets up a total of 12 symbolic characters of input: two 3-character symbolic arguments to configure the command-line parser plus another two to exercise the parsing functionality.

The test class derives from the library's `SymbolicTest` class, which provides two methods to be overridden: `setUp`, which is run once before the symbolic test starts, and `runTest`, which creates the symbolic input and can check properties. The symbolic inputs are created by calling the `getString` and `getInt` methods in the `SymbolicTest` API.

A symbolic test is executed by a symbolic test runner, which is also part of the library. The runner can work in either symbolic or replay mode. In *symbolic mode*, the runner executes inside the guest virtual machine. It creates a single instance of the test class, whose `getString` and `getInt` methods create corresponding Python objects and invoke the `make_symbolic` call to mark their memory buffers as symbolic. In *replay mode*, the runner creates one instance of the test class for each test case created by CHEF. The `getString` and `getInt` methods return the concrete input assignment of the test case.

5.2 Symbolic Execution Engine for Lua

Lua is a lightweight scripting language mainly used as an interpreter library to add scripting capabilities to software written in other languages. However, it also has a standalone interpreter and several Lua-only projects exist. We generated a symbolic execution engine for Lua based on version 5.2.2 of the Lua interpreter.

Interpreter Instrumentation Similar to Python, Lua programs are composed of one or more Lua source files, compiled into a bytecode format. The code is compiled into a set of functions that operate on a global stack of values. Each function is composed of a sequence of bytecode instructions, where each instruction is defined by an offset, opcode, and parameters. We construct the HLPC as the concatenation of the unique address of the function in the top frame and the current instruction offset being executed. The instrumentation amounts to less than 50 LoC added to the interpreter loop.

We optimized the Lua interpreter for symbolic execution by eliminating string interning. In addition, we configured the interpreter to use integer numbers instead of the default floating point, for which S2E does not support symbolic expressions. This change was easy, because it was available as a macro definition in the interpreter's configuration header.

6. Evaluation

After presenting our testing targets and methodology (§6.1), we answer the following questions:

- Is a CHEF-based symbolic execution engine effective for automated test generation (§6.2)?
- How much do CUPA and interpreter optimizations contribute to the engine's effectiveness (§6.3)?
- How efficient is the test case generation (§6.4)?
- What is the impact of each optimization technique presented in §4.2 on effectiveness (§6.5)?
- How does a generated symbolic execution engine compare to a dedicated implementation (§6.6)?

All reported experiments were performed on a 48-core 2.3 GHz AMD Opteron 6176 machine with 512 GB of

RAM, running Ubuntu 12.04. Each CHEF invocation ran on 1 CPU core and used up to 8 GB of RAM on average.

6.1 Testing Targets and Methodology

Testing Targets We evaluated the symbolic execution engines for Python and Lua on 6 Python and 5 Lua packages, respectively, including system, web, and office libraries. In total, the tested code in these packages amounts to about 12.8 KLOC. We chose the latest versions of widely used packages from the Python standard library, the Python Package Index, and the Luarocks repository. Whenever possible, we chose the pure interpreted implementation of the package over the native optimized one (e.g., the Python `simplejson` package). The first five columns of Table 3 summarize the package characteristics; LOC numbers were obtained with the `cloc` tool [1].

The reported package sizes exclude libraries, native extension modules, and the packages' own test suites. However, the packages ran in their unmodified form, using all the language features and libraries they were designed to use, including classes, built-in data structures (strings, lists, dictionaries), regular expressions, native extension modules, and reflection.

All testing targets have a significant amount of their functionality written in the interpreted language itself; we avoided targets that are just simple wrappers around native extension modules (written in C or C++) in order to focus on the effectiveness of CHEF at distilling high-level paths from low-level symbolic execution. Nevertheless, we also included libraries that depend on native extension modules. For instance, all the testing targets containing a lexing and parsing component use Python's standard regular expression library, which is implemented in C. To thoroughly test these parsers, it is important to also symbolically execute the native regular expression library. For this, the binary symbolic execution capabilities of CHEF are essential.

Methodology: Symbolic Tests For each package, we wrote a symbolic test that invokes the package's entry points with one or more symbolic strings. Figure 7 in §5.1 is an example of such a symbolic test.

Each symbolic test ran for 30 minutes within CHEF, after which we replayed the collected high-level tests on the host machine, in a vanilla Python/Lua environment, to confirm test results and measure line coverage. To compensate for the randomness in the state selection strategies, we repeated each experiment 15 times. In each graph we present average values and error margins as +/- one standard deviation.

For our experiments, we did not use explicit specifications, but relied on generic checks for finding common programming mistakes. For both Python and Lua, we checked for interpreter crashes and potential hangs (infinite loops). For Python—which, unlike Lua, has an exception mechanism—we also flagged whenever a test case led to unspecified exceptions being thrown. In general, one could find application-specific types of bugs by adding specifications in the form of assertions, as in normal unit tests.

Methodology: Coverage Measurement Line or statement coverage remains widely used, even though its meaningfulness as a metric for test quality is disputed. We measure and report line coverage to give a sense of what users can expect from a test suite generated fully automatically by a symbolic execution engine based on CHEF. For Python, we rely on the popular `coverage` package, and for Lua we use the `luacov` package.

Since our prototype only supports strings and integers as symbolic program inputs, we count only the lines of code that can be reached using such inputs. We report this number as "coverable LOC" in the fifth column of Table 3, and use it in our experiments as a baseline for what such a symbolic execution engine could theoretically cover directly. For example, for the `simplejson` library, this includes only code that decodes JSON-encoded strings, not code that takes a JSON object and encodes it into a string. Note that, in principle, such code could still be tested and covered by writing a more elaborate symbolic test that sets up a JSON object based on symbolic primitives [5].

6.2 Effectiveness for Automated Testing

We evaluate the effectiveness of the generated symbolic execution engines for bug detection and exception exploration.

Bug Detection The specifications we used for our experiments are application-agnostic and only check for per-path termination within a given time bound and for the absence of unrecoverable crashes. The first specification checks whether a call into the runtime returns within 60 seconds. In this way, we discovered a bug in the Lua JSON package that causes the parser to hang in an infinite loop: if the JSON string contains the `/*` or `//` strings marking the start of a comment but no matching `*/` or line terminator, the parser reaches the end of the string and continues spinning waiting for another token. This bug is interesting for two reasons: First, comments are not part of the JSON standard, and the parser accepts them only for convenience, so this is a clear case of an interpreter-specific bug. Second, JSON encodings are normally automatically generated and transmitted over the network, so they are unlikely to contain comments; traditional testing is thus likely to miss this problem. However, an attacker could launch a denial of service attack by sending a JSON object with a malformed comment.

The second implicit specification checks that a program never terminates non-gracefully, i.e., the interpreter implementation or a native extension crashes without giving the program a chance to recover through the language exception mechanisms. In our experiments, our test cases did not expose any such behavior.

Exception Exploration This scenario focuses on finding undocumented exceptions in Python code. Being memory-safe languages, crashes in Python and Lua code tend to be

Package	LOC	Type	Description	Coverable LOC	Exceptions	Hangs
Python						
argparse*	1,466	System	Command-line interface	1,174	4 / 0	—
ConfigParser*	451	System	Configuration file parser	145	1 / 0	—
HTMLParser*	623	Web	HTML parser	582	1 / 0	—
simplejson 3.10	1,087	Web	JSON format parser	315	2 / 0	—
unicodecsv 0.9.4	126	Office	CSV file parser	95	1 / 0	—
xlrd 0.9.2	7,241	Office	Microsoft Excel reader	4,914	5 / 4	—
Lua						
cliargs 2.1-2	370	System	Command-line interface	273	—	—
haml 0.2.0-1	984	Web	HTML description markup	775	—	—
sb-JSON v2007	454	Web	JSON format parser	329	—	✓
markdown 0.32	1,057	Web	Text-to-HTML conversion	673	—	—
moonscript 0.2.4-1	4,634	System	Language that compiles to Lua	3,577	—	—
TOTAL	18,493			12,852		

Table 3. Summary of testing results for the Python and Lua packages used for evaluation. Items with (*) represent standard library packages. Exception numbers indicate total / undocumented exception types discovered.

due to *unhandled exceptions* rather than bad explicit pointers. When such exceptions are not caught by the program, they propagate to the top of the stack and cause the program to be terminated prematurely. In dynamic languages, it is difficult to determine all the possible exceptions that a function can throw to the callee, because there is no language-enforced type-based API contract. Users of an API can only rely on the documentation or an inspection of the implementation. Therefore, undocumented exceptions are unlikely to be checked for in `try-except` constructs and can erroneously propagate further. They can then hurt productivity (e.g., a script that crashes just as it was about to complete a multi-TB backup job) or disrupt service (e.g., result in an HTTP 500 Internal Server Error).

We looked at all the Python exceptions triggered by the test cases generated using CHEF and classified them into *documented* and *undocumented*. The documented exceptions are either exceptions explicitly mentioned in the package documentation or common Python exceptions that are part of the standard library (e.g., `KeyError`, `ValueError`, `TypeError`). Undocumented exceptions are all the rest.

The sixth column in Table 3 summarizes our findings. We found four undocumented exceptions in `xlrd`, the largest package. These exceptions occur when parsing a Microsoft Excel file, and they are `BadZipfile`, `IndexError`, `error`, and `AssertionError`. These errors occur inside the inner components of the Excel parser, and should have either been documented or, preferably, been caught by the parser and re-raised as the user-facing `XLRDError`.

6.3 Impact of CUPA Heuristics and Interpreter Optimizations

We now analyze the impact of the CUPA heuristics (described in §3.2) and the interpreter optimizations (described in §4.2) on test generation effectiveness. Specifically, we measure the number of paths (respectively source code lines) covered by the test suite generated in 30 minutes for the packages in Table 3.

We compare the results obtained in 4 different configurations: (1) the baseline, consisting of performing random state selection while executing the unmodified interpreter, and then either use (2) the path- or coverage-optimized CUPA only, (3) the optimized interpreter only, or (4) both CUPA and the optimized interpreter. This way we measure the individual contribution of each technique, as well as their aggregate behavior.

Test Case Generation Figure 8 compares the number of test cases generated with each of the 4 CHEF configurations, using the path-optimized CUPA (§3.3). We only count the *relevant* high-level test cases, that is, each test case exercises a unique high-level path in the target Python program.

For all but one of the 11 packages (6 Python plus 5 Lua), the aggregate CUPA + interpreter optimizations performs the best, often by a significant margin over the baseline. This validates the design premises behind our techniques.

The CUPA strategy and the interpreter optimizations may interact non-linearly. In two cases (Python's `xlrd` and `simplejson`), the aggregate significantly outperforms either individual technique. These are cases where the result is better than the sum of its parts. In the other cases, the result is roughly the sum of each part, although the contribution of each part differs among targets. This is visually depicted on the log-scale graph: for each cluster, the heights of the middle bars measured from level $1\times$ roughly add up to the height of the aggregate (left) bar.

In one case (Lua's `JSON`), the aggregate performs worse on average than using the interpreter optimizations alone. Moreover, the performance of each configuration is less predictable, as shown by the large error bars. This behavior is due to the generated tests that cause the interpreter to hang,

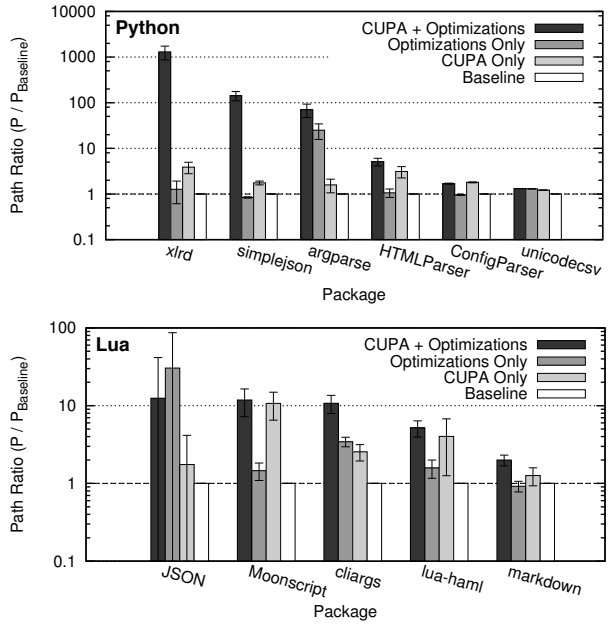

Figure 8. The number of Python and Lua test cases generated by coverage- and path-optimized CUPA relative to random state selection (logarithmic scale).

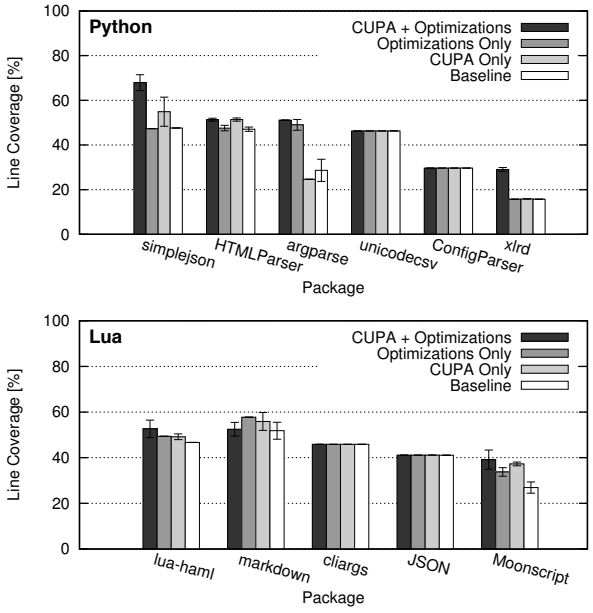

Figure 9. Line coverage for the experiments of Figure 8.

as explained in §6.2. To detect hangs, the test runs for 60 seconds before switching to another test case. This acts as a "penalty" for the configurations that find more paths leading to the hang and also skews the distribution of path execution times, since the hanging paths take significantly longer than the normal (terminating) paths.

Line Coverage Figure 9 shows the line coverage achieved by each configuration, using CUPA optimized for line coverage (§3.4). In 6 out of 11 packages, the coverage improvement is noticeable, and for Python's **`simplejson`** and **`xlrd`**, the improvements are significant (80% and 40%).

Note that these coverage improvements are obtained using basic symbolic tests that do not make assumptions about the input format. We believe that tailoring the symbolic tests to the specifics of each package could improve these results significantly.

6.4 Efficiency of Test Case Generation

We now look at the efficiency of high level test case generation. Since some low-level paths do not contribute to high-level path coverage, we evaluate the ratio of high-level tests to the total number of low-level test cases produced. We again use the same four configurations as before.

Figure 10 shows what fraction of low-level paths cover new high-level paths over time; said differently, this shows the fraction of low-level tests that turn into high-level tests. A point (t, ϕ) on the graph represents the ratio ϕ between the number of high-level paths to low-level paths explored

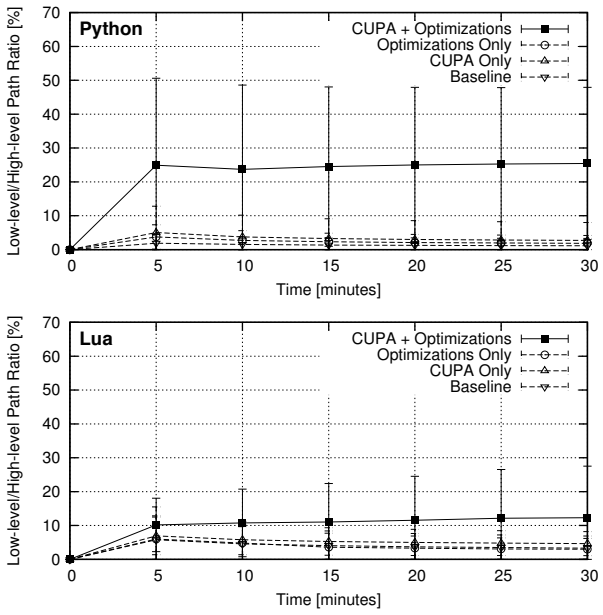

Figure 10. Average evolution in time of the fraction of low-level paths that contributed to high-level path coverage for Python and Lua.

in time t. Just like before, ϕ is averaged across all testing targets.

The aggregate configuration fares the best in the comparison, by a significant margin. Throughout the execution, the aggregate efficiency stays at about 25% for Python and 12% for Lua. This is about $10\times$, respectively $2.6\times$ higher than the best performing among the other three configurations. This

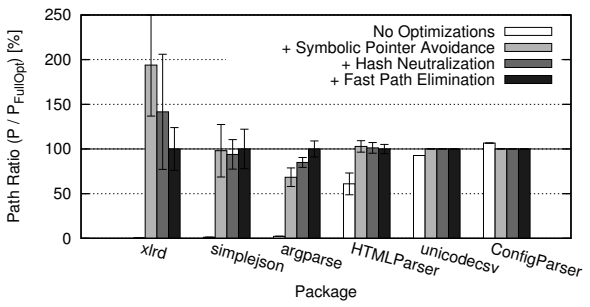

Figure 11. The contribution of interpreter optimizations for Python as number of high-level paths explored. Number of paths is relative to full optimizations (100%) for each package.

result justifies employing both CUPA and the interpreter optimizations during symbolic execution.

6.5 Breaking Down Interpreter Optimizations

We now analyze in more depth the impact of the interpreter optimizations by breaking them down into the three types mentioned in §4.2: avoiding symbolic pointers, hash neutralization, and fast-path elimination. We run again the symbolic tests for 30 minutes, using the path-optimized CUPA and four different interpreter builds, starting from the vanilla interpreter and adding the optimization types one by one. For each build and package, we count the number of high-level paths discovered by CHEF.

Figure 11 shows the results for Python. The data is normalized such that the number of high-level paths for each target reaches 100%. For 3 out of 6 packages (`simplejson`, `argparse`, and `HTMLParser`), CHEF's performance monotonically increases as more optimizations are introduced. For `unicodecsv` and `ConfigParser`, the optimizations do not bring any benefits or even hurt slightly.

However, in the case of `xlrd`, hash neutralization and fast path elimination seem to actually *hurt* symbolic execution, since the best performance is attained when only symbolic pointer avoidance is in effect. We explain this behavior by the fact that the different optimization levels cause the search strategy to explore different behaviors of the target package. `xlrd` is by far the largest Python package in our evaluation (7.2KLOC vs. the second largest of 1.4KLOC) and includes a diverse set of behaviors, each with its own performance properties.

This result suggests that, for large packages, a *portfolio* of interpreter builds with different optimizations enabled would help further increase the path coverage.

6.6 Comparison Against Hand-Made Engines

We now evaluate the trade-offs in using a symbolic execution engine generated with CHEF over building one "by hand".

Hand-Made Engines To our knowledge, no symbolic execution engine for Lua exists. For Python, we found three research tools, which we compare CHEF to. (1) CutiePy [23] is a concolic engine based on a formal model of the Python language. It uses a custom CPython interpreter to drive a concrete execution, along with updating the symbolic state according to model semantics. (2) NICE-PySE [8] is part of the NICE framework for testing OpenFlow applications. We will refer to it as NICE, for brevity. It wraps supported data types into symbolic counterparts that carry the symbolic store, and uses Python's tracing mechanisms to implement the interpretation loop fully in Python. (3) The symbolic execution engine of the scalability testing tool Commuter [11] is also entirely built in Python. Its primary purpose is the construction of models that explicitly use an API of symbolic data types.

We perform our comparison along three aspects: language features supported, implementation faithfulness, and performance. The last two aspects are evaluated only against NICE, which, besides being open source, is most compatible with our symbolic data representation (based on STP [13]).

Language Feature Support Table 4 summarizes the language feature support for CHEF, NICE, CutiePy, and Commuter, as implemented at the moment of writing. We relied on information from the respective papers in all cases and additionally on the implementation in the cases of NICE and Commuter, which are available as open source.

We distinguish engines designed to support arbitrary Python code (the "Vanilla" label) and those where the symbolic data types are an API used by model code (the "Model" label). Engines in the "Model" category essentially offer a "symbolic domain-specific language" on top of the interpreted language. CHEF, CutiePy, and NICE are "vanilla" engines, since their testing targets do not have to be aware that they are being symbolically executed. Commuter is a model-based engine, since its testing targets are bound to the symbolic API offered by the engine.

We grouped the supported language features into program state representation (the language data model and types) and manipulation (the operations on data). We divide data types into values (integers, strings and floating-point), collections (lists and dictionaries), and user-defined classes. The operations consist of data manipulation, basic control flow (e.g., branches, method calls), advanced control flow (e.g., exception handling, generators), and native method invocations (they are atomic operations at the high level). We also include in the comparison the ability to execute unsupported operations in concrete-only mode.

In a nutshell, CutiePy is able to complete correctly any execution in concrete mode by using the interpreter implementation directly. However, the symbolic semantics for each data type and native function must be explicitly provided by the developer, which makes CutiePy impractical to use with rich Python applications. NICE suffers from the additional limitation that it has to support each bytecode instruction explicitly, which makes the tool impossible to use

	CHEF	CutiePy	NICE	Commuter
Engine type	Vanilla	Vanilla	Vanilla	Model
Data types				
Integers	●	●	●	●
Strings	●	○	○	◐
Floating point	○	○	○	○
Lists and maps	●*	◐	○	●
User-defined classes	●*	○	◐	◐
Operations				
Data manipulation	●	◐	◐	◐
Basic control flow	●	●	◐	●
Advanced control flow	●	●	○	●
Native methods	●	◐	○	○

●Complete ◐Partial ○Not supported

Table 4. Language feature support comparison for CHEF and dedicated Python symbolic execution engines. Complete support with (*) refers to the internal program data flow and not to the initial symbolic variables.

beyond its target applications. Finally, Commuter provides a rich set of symbolic data types, including lists and maps, by taking advantage of Z3's extended support for arrays [12]. However, it supports only Python programs explicitly written against its API and does not handle native functions.

The engine generated by CHEF offers complete symbolic support for almost all language features. Floating point operations are supported only concretely, due to lack of support in STP, the constraint solver used by S2E. For the same reasons, the symbolic program inputs can only be integers and strings. However, all data structures are supported during the execution.

Each half or empty bullet in Table 4 implies that significant engineering effort would be required to complete a feature. While useful for their evaluation targets, NICE and CutiePy are unable to handle a complex software package that makes use of Python's many language features.

Use as Reference Implementation When the need for performance justifies investing in a dedicated engine implementation, an engine created from CHEF can serve as a reference implementation during development. One can find bugs in a symbolic execution engine by comparing its test cases with those generated by CHEF. The process can be automated by tracking the test cases generated by the target engine along the high level paths generated by CHEF to determine duplicates and missed feasible paths.

In this mode, we found a bug in the NICE implementation, which was causing it to generate redundant test cases and miss feasible paths. The bug was in the way NICE handled `if not <expr>` statements in Python, causing the engine to select for exploration the wrong branch alternate and end up along an old path. We are assisting the NICE developers in identifying and fixing any other such bugs.

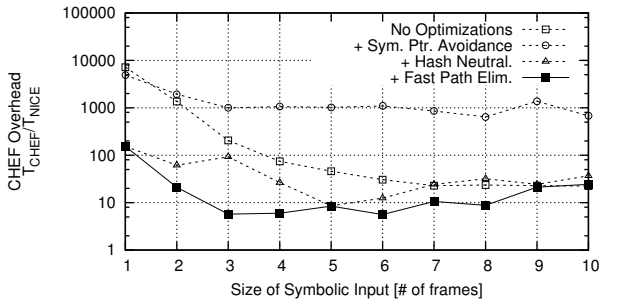

Figure 12. Average overhead of CHEF compared to NICE, computed as ratio of average per-path execution times. The average divides total tool execution time by number of high-level paths generated.

In conclusion, the experiment provides evidence that a system combining an established low-level symbolic execution engine (e.g., S2E) with a reference interpreter implementation is more robust than a symbolic execution engine built from scratch.

Performance The downside of CHEF is that the symbolic execution engines produced are slower than their hand-written equivalents. We quantify this drawback by applying CHEF to the experimental setup of NICE, consisting of an OpenFlow switch controller program that implements a MAC learning algorithm. The controller receives as input a sequence of Ethernet frames and, in response, updates its forwarding table (stored as a Python dictionary). We use symbolic tests that supply sequences of between 1 and 10 Ethernet frames, each having the MAC address and frame types marked as symbolic.

Given the small size of the controller (less than 100 LOC), the number of execution paths is relatively small, and choosing low-level paths at random quickly discovers new high-level paths. Therefore, the search strategy has no impact (in the experiments we used path-optimized CUPA). However, the interpreter optimizations are crucial, since the controller code relies heavily on the dictionary. As in §6.5, we use several interpreter builds with optimizations introduced one-by-one.

Figure 12 illustrates the overhead for each optimization configuration, as a function of number of Ethernet frames supplied. The overhead is computed as the ratio between the average execution times per high-level path of NICE and CHEF. In turn, the execution time per high-level path is computed by dividing the entire execution time of each tool by the number of paths it produced.

The performance of each optimization configuration illustrates the sources of path explosion and slowdown in the vanilla interpreter. With no optimizations, symbolic keys in the MAC dictionary cause massive path explosion due to symbolic pointers. When avoiding symbolic pointers, performance drops even more due to symbolic hash computa-

tions. This penalty is reduced up to two orders of magnitude with hash neutralization. Finally, fast path elimination reduces the forking inside string key comparisons in the dictionary.

The shape of the final performance curve (the solid line) is convex. For 1 and 2 symbolic frames, the search space is quickly exhausted and the execution time is dominated by CHEF's initialization costs, i.e., setting up the symbolic VM and executing the interpreter initialization inside the guest. This results in an execution overhead as high as 120×. For more symbolic frames, the initialization cost is amortized, and the overhead goes below 5×. However, as the number of frames increases, so does the length of the execution paths and the size of the path constraints, which deepens the gap between CHEF's low-level reasoning and NICE's higher level abstractions. For 10 symbolic frames, the overhead is around 40×.

Despite CHEF's performance penalty, the alternative of writing an engine by hand is daunting. It involves developing explicit models that, for a language like Python, are expensive, error-prone, and require continuous adjustments as the language evolves. Where performance is crucial, a hand-written engine is superior; however, we believe that CHEF is a good match in many cases.

7. Related Work

To the best of our knowledge, we are the first to use symbolic execution on an interpreter to symbolically execute a program written in the target interpreted language. However, there has been work on writing dedicated symbolic execution engines for interpreted languages directly. Beside Python engines, the Kudzu [24] symbolic execution engine for JavaScript was used to detect code injection vulnerabilities. It relies on an intermediate representation of JavaScript that it directly executes symbolically. Apollo [2] is another engine targeting PHP code to detect runtime and HTML errors, while Ardilla [17] uses this system to discover SQL injection and cross-site scripting attacks in PHP.

The general area of automated software testing has a rich body of published literature. We highlight here only the closest concepts. Symbolic tests are closely related to the idea of parameterized unit tests [26]. These extend regular unit tests with parameters marked as symbolic inputs during symbolic execution. QuickCheck [10] allows writing specifications in Haskell, which again share their basic concept with symbolic tests, and tries to falsify them using random testing. Symbolic execution can offer an alternative to random testing in evaluating QuickCheck test specifications.

8. Conclusion

Implementing and maintaining a symbolic execution engine is a significant engineering effort. It is particularly hard for interpreted dynamic languages, due to their rich semantics, rapid evolution, and lack of precise specifications. Our system CHEF provides an engine platform that is instantiated with a language interpreter, which implicity defines the complete language semantics, and results in a correct and theoretically complete symbolic execution engine for the language. A language-agnostic strategy for selecting paths to explore in the interpreter allows the generated engine to systematically explore and test code in the target language (including possible native calls) effectively and efficiently. CHEF is available at `http://dslab.epfl.ch/proj/chef`.

Acknowledgments

We thank the reviewers and Jonas Wagner for their valuable feedback on our paper. We thank Marco Canini, Peter Perešíni, and Daniele Venzano for their generous help with NICE. We are grateful to the European Research Council (StG #278656) and to Google for supporting our work.

References

[1] Al Danial. Cloc. `http://cloc.sourceforge.net/`.

[2] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *Intl. Symp. on Software Testing and Analysis*, 2008.

[3] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillère. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *Computer Software & Applications Conference*, 2013.

[4] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. Technical Report MSR-TR-2012-55, Microsoft Research, 2012.

[5] S. Bucur, J. Kinder, and G. Candea. Making automated testing of cloud applications an integral component of PaaS. In *Proc. 4th Asia-Pacific Workshop on Systems (APSYS 2013)*. USENIX, 2013.

[6] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Intl. Conf. on Automated Software Engineering*, 2008.

[7] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Sys. Design and Implem.*, 2008.

[8] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE way to test openflow applications. In *Symp. on Networked Systems Design and Implem.*, 2012.

[9] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[10] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN International Conference on Functional Programming*, 2000.

[11] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Symp. on Operating Systems Principles*, 2013.

[12] L. de Moura and N. Bjorner. Generalized, efficient array decision procedures. In *Intl. Conf. on Formal Methods in Computer-Aided Design*, 2009.

[13] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Intl. Conf. on Computer Aided Verification*, 2007.

[14] P. Godefroid. Compositional dynamic test generation. In *Symp. on Principles of Programming Languages*, 2007.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Intl. Conf. on Programming Language Design and Implem.*, 2005.

[16] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symp.*, 2008.

[17] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Intl. Conf. on Software Engineering*, 2009.

[18] R. S. King. The top 10 programming languages. *IEEE Spectrum*, 48 (10):84, 2011.

[19] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *USENIX Annual Technical Conf.*, 2010.

[20] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Intl. Conf. on Programming Language Design and Implem.*, 2012.

[21] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. on Code Generation and Optimization*, 2004.

[22] *The Python Language Reference*. Python Software Foundation. `http://docs.python.org/3/reference/`.

[23] S. Sapra, M. Minea, S. Chaki, A. Gurfinkel, and E. M. Clarke. Finding errors in python programs using dynamic symbolic execution. In *Intl. Conf. on Testing Software and Systems*, 2013.

[24] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *IEEE Symp. on Security and Privacy*, 2010.

[25] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Intl. Conf. on Information Systems Security*, 2008.

[26] N. Tillmann and W. Schulte. Parameterized unit tests. In *Symp. on the Foundations of Software Eng.*, 2005.

[27] J. Wagner, V. Kuznetsov, and G. Candea. -OVERIFY: Optimizing programs for fast verification. In *Workshop on Hot Topics in Operating Systems*, 2013.

[28] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Intl. Conf. on Dependable Systems and Networks*, 2009.

[29] C. Zamfir and G. Candea. Execution synthesis: A technique for automated debugging. In *ACM EuroSys European Conf. on Computer Systems*, 2010.