Conpy: Concolic Execution Engine for Python Applications

Ting Chen[1], Xiao-song Zhang[1], Rui-dong Chen[1], Bo Yang[1], and Yang Bai[2]

[1] School of Computer Science & Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China
[2] No.30 Research Institute, China Electronics Technology Group Corporation (CETC), Chengdu 618841, China
chenting19870201@163.com

Abstract. Concolic execution has become a promising technique for program analysis in recent years, whereas it rarely applies to Python applications. In this work, we propose a concolic execution engine for Python applications named Conpy. Conpy is easy to deploy since it is written in pure Python and it is not dependent on any third-party tools. Conpy is also easy to use. Anyone with basic knowledge of Python and concolic execution can quickly get start with Conpy. Besides, Conpy works in low level and produces human-readable reports which facilitate subsequent analysis. We then make an elaborate performance testing on Conpy. Results show that the overhead of Conpy is acceptable, that is to say, less than one order of magnitude in most cases.

Keywords: Python, concolic execution, easy to deploy, easy to use, low overhead.

1 Introduction

Concolic execution, or dynamic symbolic execution, which is a variation of traditional symbolic execution proposed in 1970s [1], is now becoming a hot technique for program analysis. As the name implies, concolic execution combines concrete execution which runs the programs under analysis (PUA) concretely with symbolic execution which marks symbols, tracks symbols and produces path condition in the meanwhile. So far, concolic execution has applied to software testing, software bugs finding and malware analysis. A number of concolic execution tools have been proposed recently, such as SAGE [2], Pex [3], KLEE [4], DART [5], CUTE [6], Fuzzgrind [7], Catchconv [8], S2E [9], Splat [10], TaintScope [11], BitBlaze [12], CREST [13], JPF-SE [14] as well as our previous tools SMAFE [15] and SEVE [16].

However, rare tools can handle Python applications. Python was often used as a scripting language for web applications, but now it is widely used by large organizations including Google, Yahoo!, NASA for its high productivity [17]. So it is meaningful to design and implement a concolic execution tool for Python applications.

X.-h. Sun et al. (Eds.): ICA3PP 2014, Part II, LNCS 8631, pp. 150–163, 2014.

We have two major contributions in this work. The first, we design and implement a concolic execution engine for Python applications named Conpy. Conpy is written in pure Python and it does not depend on any third-party tools, so it can be released with Python's codebase as a module. Besides, Conpy is easy to use. Typically, the only effort for users is to specify symbol sources, such as the inputs read from files, data comes from Internet, return values of any functions. Conpy runs in low level, so it is able to produce reports containing low-level information, which obviously benefit subsequent analysis. Furthermore, the reports generated by Conpy are human-friendly. So experts can analyze the reports manually or interpret the reports by any third-party tools.

The second contribution is that we test the performance (i.e. overhead) of Conpy extensively. To our knowledge, the overhead issue of existing concolic execution tools has not been studied in depth. Current concolic execution tools are employed in off-line analysis, such as automated test generation, software bugs finding etc., rather than on-line detection or protection. So overhead is not so critical for current usage. The results from performance testing are promising that in most cases the overhead is less than one order of magnitude.

2 Related Work

This section briefly reviews the design and implementation of existing concolic execution tools in order to outline the differences between them and our Conpy. We try to present some typical tools which have got high attention, but we do not intend to present all of them. Readers who are interested in current concolic execution tools can refer to a recent survey [18].

SAGE [2] is proposed by Microsoft Corporation which concolically executes Windows binaries. SAGE is built on top of a trace replay framework, named TruScan. The trace files consumed by TruScan are produced by iDNA framework. So concolic execution in SAGE is according to the following process: (1) executes PUA concretely and gathers trace files by iDNA. (2) executes trace files symbolically with the help of TruScan. The paper [2] reports SAGE is rather slow and it gives an example to illustrate how slow SAGE is. The symbolic execution of Media 2 with wff-3 takes 25 minutes 30 seconds, while concrete execution costs only several seconds.

DART [5], CUTE [6], Splat [10] and CREST [13] are instrumented by a source-to-source translator, named CIL. New source files will be produced by CIL, including original source which performs concrete execution and instrumented functions which perform symbolic execution. Those tools above can handle C programs only when source code is available since CIL requires C source code. JPF-SE [14] is based on Java Pathfinder which applies a similar approach: it utilizes a Java source-to-source translation tool for instrumentation.

Fuzzgrind [7] and Catchconv [8] employ Valgrind to instrument PUA dynamically. That is, the function of symbolic execution is instrumented when PUA is running. TaintScope [11], as well as our previous tools SMAFE [15] and

SEVE [16] do a similar work based on another dynamic instrumentation tool, Pin. Similarly, Minesweeper [19] enhances Qemu with dynamic binary instrumentation. Those tools which are built on dynamic binary instrumentation can handle unmodified binary PUA with the cost of very high overhead. For example, the average overhead introduced by Pin and Valgrind is 2.5 times and 8.3 times respectively even if a simple task for basic-block counting is instrumented. So it is not surprising that orders of magnitude overhead will be incurred by those dynamic-instrumentation-based concolic execution tools.

KLEE [4] itself acts as an interpreter to concolically execute the bytecode produced by LLVM. KLEE runs below PUA and it is in charge of the execution of PUA. KLEE does not instrument PUA in static or dynamic way. Instead, KLEE directly interprets bytecode and maps bytecode to constraints. S2E [9] reuses the symbolic execution engine of KLEE to handle both user-mode and kernel-mode binaries. To translate x86 instructions to the bytecode that KLEE's symbolic execution engine can understand, S2E proposes an x86-to-LLVM backend for QEMU on where the guest OS and S2E run. As mentioned in paper [9], the overhead of S2E in symbolic mode is $\sim$78 times.

Pex [3] adopts a dynamic-instrumentation-based method to concolically execute .NET code. It instruments by .NET profiling API and it enhances .NET virtual machine with symbolic execution. BitBlaze [12] and BitScope [20] implement a plugin for TEMU to mark and track symbols. As PUA runs in TEMU, so those tools have the privilege to interpret the execution of PUA in symbolic mode.

Conpy employs a different design with existing concolic execution tools. Unlike DART, CUTE, Splat, CREST, JPF-SE, no source-to-source translators are required. Compared to Fuzzgrind, Catchconv, TaintScope, SMAFE, SEVE etc., Conpy does not depend on dynamic instrumentation tools. Besides, Conpy does not require virtue machines like Qemu and Temu. Moreover, Conpy does not behave as an interpreter like KLEE. Actually, Conpy does not care about (or even be aware of) the executions which do not involve symbolic computations. Also, in Conpy, symbolic execution is along with concrete execution, so it is different from the replay scheme of SAGE.

This work conducts experiments to measure the overhead of Conpy. We find that only two (i.e. SAGE [2] and S2E [9]) existing concolic execution tools briefly report their overhead that is high. Concretely speaking, paper [2] presents an example to show SAGE's high overhead and paper [9] claims the overhead of S2E in symbolic mode is $\sim$78 times. The overhead evaluation of the two tools considers the whole process of test data generation such as concolic execution, constraint solving, path exploration and so on. Differently, this work measures the overhead of concolic execution only which includes concrete execution and symbolic execution (e.g. symbol marking, symbol tracking, generation of path conditions).

3 Design

3.1 Principles

At first, we present the design principles of Conpy, and then describe the design scheme. The design principles are actually the goals we want to achieve.

1. Conpy should be easy to deploy. That is to say, it is able to run on mainstream hardware platforms and software platforms. Besides, this tool would better not depend on any special third-party tools. Otherwise, if the tool can run on special situations only, its practicability will be impaired.
2. Conpy should be easy to use. The reason is obvious that too much pre-knowledge and complex usage will hinder users from getting start with the tool.
3. Conpy runs in low level and produces reports with low-level information. Even if it is not always so, low-level information is usually richer and more interesting than high-level information. For instance, if Conpy runs in the string level, it can discern whether the two strings are equal or not. But if it runs in the byte level, additional information can be got: if the two strings are not equal, Conpy can tell users which bytes are different.

   Besides, high-level concolic execution tool may produce more false alarms than the low-level counterparts. For example, assume a string $s1$ is a symbol, if the concolic execution tool is now in the string level, the new string $s2$ which is the concatenation of $s1$ and a non-symbol string $s3$ should be symbolized. So consider the string $s4$, which is computed as $s2[len(s2)-len(s3):]$, $s4$ should be symbolized since $s2$ is a symbol. However it is incorrect because $s4$ equals to $s3$ in essence. On the contrary, if the concolic execution tool runs in the byte level, $s4$ will be recognized accurately.

   Additionally, low-level concolic execution facilitates tracking symbols among different data types. For example, in statement $s =$ "$\%i$"$\%x$, x is a symbolized integer. After the execution of this statement, Conpy will map the symbol linked with x to each byte of the string s, thus string s will be symbolized.
4. The overhead of Conpy should be acceptable. Even if current concolic execution tools are applied in off-line analysis, overly high overhead may make them impracticable.

3.2 Scheme

The design of Conpy is based on the claim "everything in Python is an object" [21]. Even the primitive types such as "int", "float", "bool" are objects, which is different from traditional programming languages such as C, C++. The core idea of Conpy consists of extending the Python's objects with an additional attribute termed by *symbol* or *symbol_set* indicating the symbol(s) linked with the objects and enhancing Python's functions with the abilities to track symbols and produce path conditions.

Whether an object links with a symbol or a symbol set depending on whether the object is a primitive type or a container. For example, an integer which is a

primitive type links with a symbol but a string that is a container is associated with a symbol set. Each item in the symbol set indicates whether the corresponding item in the container is a symbol or not. Put it another way, if an item in the symbol set is empty, the associated item in the container is not a symbol even if the container itself is already symbolized. In this way, Conpy is able to run in primitive level rather than much higher container level.

Consider the following example, assume a string with length three is symbolized as $s1$, the symbol set of $s1$ is something like [‘$b1$’, ‘$b2$’, ‘$b3$’] indicating the three bytes of the string links with three symbols $b1$, $b2$ and $b3$ respectively. If a new string $s2$ is the concatenation of $s1$ and a constant string, say ‘abc’, the symbol set of $s2$ should be [‘$b1$’, ‘$b2$’, ‘$b3$’, ‘’, ‘’, ‘’]. Therefore, by looking up the symbol set, Conpy is aware that the last three bytes of $s2$ are actually not dependent on symbols.

The concolic execution of Conpy typically consists of three procedures: marking symbol sources, tracking symbols and producing path conditions. The last two steps are fully automated, only the first one requires human intervention. To mark symbol sources, users is asked to specify which variables are of interested. This step is made by simply calling the interface *symbolize* provided by Conpy. To facilitate users, Conpy can run in a fully automated mode which marks all inputs read from the environment as symbols.

The general principle of symbol tracking is: mapping the effects of statements to the symbols of destination variables if the statements involve symbolic computations. The mapping process should maintain the semantics of the statements. For example, given a statement $y = x + 10$, assuming before its execution, y and x have been symbolized as $i0$ and $i1$ respectively. After its execution, the symbol associated with y should be updated as $i1 + 10$.

Constraints should be produced when the comparisons which involve symbolic computations are made. Like symbolic expressions, constraints should maintain the semantics of comparison statements. A critical step of producing constraints is to discern which statements can produce constraints. In many cases, the statements for comparison are obvious, such as $x == y, x >= y, x! = 0$.

But in some cases, comparisons are implicit. For instance, if Conpy symbolizes a string by invoking the interface *symbolize*, it should produce a constraint like $s1 == b0.‘a’.b3$. The constraint contains the following information: the string is symbolized as $s1$; its first byte corresponds to a symbol $b0$; its second byte is a constant ‘a’; its last byte links with a symbol $b3$. Let’s consider another example, *find* is a build-in function of the *str* type which returns the lowest index where the substring is found. Conpy extends *find* with constraint generation if comparisons in *find* involve symbols.

To get low-level path conditions, Conpy runs in primitive type level. That’s to say, Conpy treats a primitive type as an atom which cannot be broken down. While executing a container concolically, Conpy dives into the container and symbolizes primitive variables in the container. For example, Conpy symbolizes each byte of a string. Given a list of integers, Conpy treats each integer of the list as a symbol.

Low overhead can be achieved through our design since only the symbolized variables are manipulated by Conpy. In other words, Conpy does not care about or even not be aware of non-symbol variables. So actually the computations without symbols run in native mode which incurs no overhead. Experiments validate our claim that the overhead of a symbol-intensive program is as high as 46 times, but the overhead of a symbol-non-intensive program is only about 80%.

4 Implementation

This section presents the implementation details as well as some critical code of Conpy. As a representative of primitive type, we present the handling of *int* in Section 4.1. The other primitive types such as *float, bool* are handled similarly.

Then we show the concolic execution of *str* in Section 4.2, which is a frequently-used container in Python. We handle the other containers such as *list*, *tuple* in a similar way. In fact, we find that the handling of *str* is more intricate than the other containers. One obvious reason is that *str* provides so many build-in functions which require extensions for concolic execution. Additionally, the atomic items of *list*, *tuple* etc. may be the primitive types we have already handled such as *int, float, bool*. So considerable development effort can be saved. However, the atomic items of *str* is still *str*. So we have to manipulate the *str* type from scratch.

Finally, we show how symbols spread from the *int* type to the *str* type and back in Section 4.3.

4.1 Manipulation of *int*

To mark new integer symbols, users need to invoke the interface *symbolize* with the argument being an integer. After discerning the type of the argument, the corresponding routine *symbolize_int* is called. The function *symbolize_int* is defined as:

```
1  def symbolize_int(val):
2    sym_int = symbol_int(val)
3    sym_int.symbol = symbol(0)
4    return sym_int
```

The integer is passed as the argument *val* in line 1. In line 2, a new object of type *symbolize_int* is created. Actually, the class *symbolize_int* inherits from the primitive type *int*. So all attributes and build-in functions of *int* are also available in *symbolize_int*. Line 3 links the attribute *symbol* of object *symbolize_int* with an object of class *symbol*. Finally the object *symbolize_int* is returned. The code of initializing an object of class *symbol* is:

```
1  class symbol:
2    sym_num = 0
3    def __init__(self, f_atom):
4      if f_atom==0: #atomic symbol
5        self.sign = sign_none
6        self.atom= 'i' + str(symbol.
             sym_num)
7        symbol.sym_num += 1
8        self.operand_num = 0
9        self.operands = []
```

The variable *sym_num* which is defined in line 2 denotes the number of symbols. Once an atomic symbol is created, the variable will increase by 1 (line 7). If a new symbol is created from existing symbols (i.e. the predicate in line 4

evaluates as false), the following statements will not be executed. Conpy organizes a symbol as a tree in memory. As an atomic symbol, the sign of the symbol is initialized as *sign_none* (line 5) and its operands are empty (line 8 and line 9).

To track symbols, Conpy overrides the build-in functions which are able to spread symbols. We present the implementation of *__add__* as an example.

```
1  def __add__(first, second):
2    res = int.__add__(first, second)
3    if isinstance(first, symbol_int):
4      if isinstance(second, symbol_int):
5        sym = symbol.__add__(first.symbol, second.symbol)
6      else:
7        sym = symbol.__add__(first.symbol, second)
8      add_int = symbol_int(res)
9      add_int.symbol = sym
10   else: pass
11   return add_int
```

When a statement like $x+y$ executes, the build-in function *__add__* or *__radd__* will be called depending on which operands (x or y or both) are symbols. Specifically, if the first operand is a symbol (no matter whether the second operand is a symbol or not), the function *__add__* will be invoked. In cases when the first operand is not a symbol but the second is, in turn the function *__radd__* will be called. Note that if neither of them are symbols, original function of class *int* will be called, thus no additional overhead will be introduced in this case.

The original function *__add__* of class *int* is invoked to compute the concrete outcome of the statement (line 2). Then the integer *res* is symbolized as *add_int* in line 8. The symbol linked with *add_int* is created by invoking the function *__add__* of class *symbol*. Below is the core code of *symbol.__add__*.

```
1  def __add__(sym1, sym2):
2    add_sym = symbol(1)
3    add_sym.sign = sign_plus
4    add_sym.operand_num = 2
5    add_sym.operands = []
6    add_sym.operands.append(sym1)
7    add_sym.operands.append(sym2)
8    return add_sym
```

A new symbol is created in line 2. As the new symbol is the plus of existing symbols, the argument of class *symbol* should be 1. The sign of the symbol is defined as *sign_plus* (line 3) denoting the symbol results from a plus operation. The symbol has two operands (line 4) and each of them is a symbol or a concrete value. The two operands are appended in the set *operands* as two subtrees of symbol *add_sym* (line 6 and line 7). Finally, the symbol is returned.

Conpy will produce constraints when the comparisons which involve symbolic computations are executed. Consider the following example, when executes the statement $x == y$, the build-in function *__eq__* will be called. So Conpy extends the function with the ability to produce constraints as follows:

```
1  def __eq__(val1, val2):
2    res = int.__eq__(val1, val2)
3    if isinstance(val1,symbol_int)& isinstance(val2, symbol_int):
4      print_sym(val1.symbol)
5      if res == 1:
6        print('==', end = '')
7      else:
8        print('!=', end = '')
9      print_sym(val2.symbol)
10   elif: # code for the other situations
11   return res
```

The concrete outcome of the comparison is got by invoking original function of class *int* in line 2. If both *val*1 and *val*2 are symbols, the statements from line 4 to line 9 will be executed. The other situations are handled similarly, so related code does not present here. The symbols of *val*1 and *val*2 are printed by the function *print_sym*. The function *print_sym* is actually a recursive function which traverses the trees where store symbols. Naturally, depending on the outcome of comparison, '==' or '!=' is printed.

4.2 Manipulation of *str*

To mark a string as a symbol, users invoke the interface *symbolize*. Inside the this function, routine *symbolize_str* is invoked as the argument being a string. Function *symbolize_str* works similar with the function *symbolize_int* except two aspects. First, a symbol set rather than a symbol links with the symbolized string. The symbol set is created by invoking function *create_sym* of class *symbolize_str* which inherits from the build-in class *str*. Related code is:

```
1  def create_sym(s):
2    sym_set = []
3    for i in range(0, len(s)):
4      sym = symbol()
5      sym.sign = sign_none
6      sym.atom = 'b' + str(symbol_str.sym_num)
7      sym_set.append(sym)
8      symbol_str.sym_num += 1
9    return sym_set
```

Conpy links each byte of the string with a symbol through the loop from statement 3 to statement 8. Second, a constraint denoting the string is the concatenation of its bytes is produced through the function *print_create_cons* of class *symbol*.

Conpy tracks the propagation of symbols by overriding build-in functions of class *str*. Consider the example, given a symbolized string s with its symbol being $b0.b1.b2$, after the execution of the statement $s = s * 2$, the symbol of s should be updated as $b0.b1.b2.b0.b1.b2$. To this purpose, Conpy extends the build-in functions *__mul__* and *__rmul__* with the ability to symbolic execution. We present *__mul__* here.

```
1  def __mul__(s, count):
2    res = str.__mul__(s, count)
3    sym = symbol_str(res)
4    sym.name = 's' + str(symbol_str.str_num)
5    symbol_str.str_num += 1
6    ln = len(s.sym_set)
7    sym.sym_set=[0]*ln*count
8    for i in range(0, count):
9      sym.sym_set[i*ln:(i+1)*ln]=s.sym_set
10   symbol.print_create_cons(sym)
11   return sym
```

The outcome of multiplication is computed by invoking original function *__mul__* of class *str* (line 2). The symbol set of symbolized string is copied from the symbol set of argument s by *count* times (line 8 and line 9).

When statements like $x == y, x > y, x! = y$ execute, Conpy will produce constraints if those statements involve symbolic computations. Conpy handles those

statements similarly with the manipulations of symbolized integers except that the comparison of two strings usually produces a number of constraints rather than only one. Conpy handles *find, rfind, __contains__, index etc.* specially because those functions can produce constraints besides their original return values. We do not show associated code here since the code is relatively long. In short, the basic idea is to use symbolic strings to simulate operations of concrete strings.

4.3 Symbol Propagation between *str* and *int*

Conpy runs in low level making it is able to spread symbols among different types. This section shows how symbols propagate from *int* type to *str* type through function *chr* and back through function *ord*. The basic idea is to extend functions *chr* and *ord* with the abilities to spread symbols and produce constraints. Look at the code below:

```
1  def symbol_func(org):
2    def inner(*args, **kwargs):
3      r = org(*args, **kwargs)
4      if org.__name__ == 'ord':
5        c = args[0]
6        if isinstance(c, symbol_str):
7        res = symbolize(r)
8        print_sym(res.symbol)
9        print('==', end = '')
10       print_sym(c.sym_set[0])
11       return res
12     elif org.__name__ == 'chr':
13       i = args[0]
14       if isinstance(i, symbol_int):
15         res = symbolize(r)
16         print_sym(res.sym_set[0])
17         print('==', end = '')
18         print_sym(i.symbol)
19         return res
20     return r
21   return inner
22 ord = symbol_func(ord)
23 chr = symbol_func(chr)
```

Original functions *chr* and *ord* are enhanced by invoking function *symbol_func* in line 22 and line 23. In *symbol_func*, original function is invoked first to get the concrete return value *r* (line 3). If the function is *ord* , the first item in *args* should be the argument passed to *ord* and then the item is assigned to a variable *c* for convenience (line 5). If *c* is symbolized, the return value *r* should be symbolized accordingly (line 7). After that, a constraint indicating the correlation between the symbol of *res* and the symbol of *c* is produced (lines 8 – 10). Finally, the symbolized integer *res* is returned. The handling of function *chr* is similar (lines 13 – 19).

5 Experiments

5.1 Setup

Our experiments have two goals. The first is to validate the ability of Conpy to produce path conditions. The second is to evaluate the performance overhead of Conpy. We first present test environment. CPU is an Intel Core i7-2760QM with the frequency of 2.4GHz. The capacity of main memory is 8GB. Operating system is 64-bit Windows 7 Home Premium. The version of Python installed on my computer is 3.3.0.

We select ten programs which are all standard modules of Python as benchmarks. The reason for choosing those programs lies in that those modules are frequently invoked by any other Python programs making experimental results more valuable. Results as well as corresponding test harnesses are presented in the following section.

5.2 Results

We first summarize test results in table 1 and then detail each tested program with its test harness. The second column of table 1 shows the length of path condition of each tested program. To measure overheads accurately, we run some of tested programs multiple times. So we get very long path conditions. The last column gives the overhead of each tested program. We can observe that seven out of ten tested programs slow down by less than one order of magnitude. The highest overhead is about 46 times when testing *heapq* module. The overhead can be as low as 80% when testing *imghdr* program. Actually, we can expect even lower overhead after further optimizations of our implementation.

Table 1. Test results

Program	Length of PC	Overhead
heapq	142825	46
calendar	1749090	5.1
random	34662	2.7
bisect	237987	4.1
html.parser	237000	12.1
re	140042	5.1
mimetypes	62000	6.4
urllib.parse	280114	27.6
imghdr	301000	0.8
sndhdr	30000	2.9

heapq. The tested program is *Lib/heapq.py* which provides an implementation of heap queue algorithm. Test harness is shown below. The code for importing Conpy, producing random integers, outputting the statistics of performance overhead *etc.* is omitted.

```
1  #ln = symbolize(ln)
2  h = []
3  for v in ln:
4    heappush(h, v)
5  heappop(h) for i in range(len(h))
```

The list *ln* contains 10000 random integers. After sorting, the result is stored in the list *h*. By uncommenting line 1, the program will run concolically, otherwise, it will run concretely. Time cost for concrete execution is about 6.7ms while the time for concolic execution is about 313.8ms which is roundly **46** times longer than concrete execution. Path condition produced by Conpy is as long as 142825 denoting there are 142825 constraints in the path condition.

calendar. It is a program (*Lib/calendar.py*) allows to output calendars like the Unix Lib *cal* program. Test harness is given below.

```
1  for year in range(1000, 3000):
2    for month in range(1, 13):
3      #year = symbolize(year)
4      #month = symbolize(month)
5      matrix = monthcalendar(year, month)
```

As variable *year* ranges from 1000 to 3000 and variable *month* ranges from 1 to 13, function *monthcalendar* will be called 24000 times. The reason for running *monthcalendar* so many times is that it facilitates to evaluate performance overhead. Time for concrete execution is about 744ms. By uncommenting line 3 and line 4, the program runs concolically. Consequently, time consumption increases to 4538.3ms (i.e. overhead is about **5.1** times). Path condition consists of 1749090 constraints.

random. It is a program (*Lib/random.py*) implements pseudo-random number generators for various distributions. Below presents test harness.

```
1  for i in range(0, 10000):
2    #i = symbolize(i)
3    x = randrange(i)
```

Running the above program, 10000 random integers are generated in about 60.8ms. In concolic mode, the program costs about 223.8ms indicating overhead is about **2.7** times. The length of path condition is 34662.

bisect. It is a program (*Lib/bisect.py*) provides support to maintain a list in sorted order without having to sort the list after each insertion. This module is termed by *bisect* because it uses a basic bisection algorithm to do its work. Below is the test harness.

```
1  for i in range(0, 10000):
2    #r = symbolize(r)
3    bisect(list, r)
4    insort(list, r)
```

The variable r is a random integer. Time for concrete execution is 121.2ms. Concolic execution costs about 612.7ms which is **4.1** times longer than concrete execution. Produced path condition contains as many as 237987 constraints.

html.parser. The module (*Lib/html/parser.py*) defines a class *HTMLParser* which serves as the basis to parse text files formatted in *HTML* or *XHTML*. Here is the test harness.

```
1  for i in range(0, 1000):
2    parser = HTMLParser(strict = False)
3    s = ' <html><head><title>Test</title></head><body><h1>Parse me!</h1></
         body></html> '
4    #s = symbolize(s)
5    parser.feed(s)
```

To evaluate the performance overhead of Conpy accurately, the parsing process repeats for 1000 times. The parsed *html* text is given in line 3 which will be symbolized in line 4. In concrete mode, the program costs about 133.2ms. Switching to concolic mode, time consumption rises to 1748.2ms. In other words, the overhead incurred by Conpy is about **12.1** times. The length of path condition is 237000.

re. This module provides regular expression matching operations. Test harness is shown as following.

```
1  for i in range(0, 10000):
2    s1 = ' (?<=abc)def '
3    s2 = ' abcdef '
4    #s1 = symbolize(s1)
5    s2 = symbolize(s2)
6    m = re.search(s1, s2)
```

Function *search* provided by *re* module is used to look for a location where the regular expression (as shown in line 2) produce a match, and return a

corresponding match object. Search process repeats for 10000 times so as to evaluate Conpy's overhead adequately. Time cost by concrete execution is about 159.6ms. The program runs concolically by uncommenting line 4 and line 5. Time consumption in concolic mode is 981ms. So the overhead of Conpy is about **5.1** times. The number of constraints produced by Conpy is as many as 140042.

mimetypes. The *mimetypes* module (*Lib/mimetypes.py*) converts between a filename or a *URL* and the *MIME* type associated with the filename extension. Two functions *guess_type* and *guess_extension* are tested in the following test harness.

```
1  minetypes.init()
2  for i in range(0, 1000):
3    fname = '/ct/ycg/f.tgz'
4    #fname = symbolize(fname)
5    res = mimetypes.guess_type(fname)
6    ty = res[0]
7    #ty = symbolize(ty)
8    mimetypes.guess_extension(ty)
```

To evaluate performance overhead reliably, test progress repeats 1000 times. Concrete execution costs 70.4ms, while concolic execution spends about 523ms which is **6.4** times longer. The path condition produced by Conpy consists of 62000 constraints.

urllib.parse. This module (*Lib/urllib/parse.py*) defines a standard interface to break *URL* strings up in components, to combine the components back into a *URL* string, and to convert a "relative *URL*" to an absolute *URL* given a "base *URL*". We test function *urlparse* through the following harness.

```
1  for i in range(0, 5000):
2    url = ' http://www.uestc.edu.cn/index.html '
3    #url = symbolize(url)
4    urlparse(url)
```

Time cost for concrete execution is about 45.7ms. By uncommenting line 3, the program runs concolically which spends 1307.4ms. Overhead is about **27.6** times. The length of path condition is 280114.

imghdr. This module (*Lib/imghdr.py*) determines the type of image contained in a file or byte stream. We reuse the test harness included in the source *imghdr.py* by only a few modifications. For example, we insert a line of code *h = symbolize(h)* after line 13 to symbolize the data read from input file. Test harness recognizes whether the type of files under current path is image. We repeat test process for 1000 times to get accurate information of performance overhead. The result is that concrete execution spends 5326.5ms, while concolic execution costs about 9552ms. That is to say, the overhead of Conpy is about **80%**. The path condition consists of 301000 constraints.

sndhdr. The sndhdr module (*Lib/sndhdr.py*) provides utility functions which attempt to determine the type of sound data which is in a file. We reuse the test harness in the source *sndhdr.py*. Test harness recognizes whether the files under current path are sound files or not. We just make only a few modifications to run the program concolically. For example, we add a line of code *h = symbolize(h)* after line 44. Additionally, the test process repeats 100 times in order to evaluate overhead. Time consumption for concrete execution is about 663.2ms, while the time for concolic execution is 2591.7ms. So the overhead incurred by Conpy is about **2.9** times. The generated path condition contains 30000 constraints.

6 Conclusion

Concolic execution is a promising technique for program analysis. But so far we have not observed any concolic execution tools for Python applications. In this work, we propose a concolic execution engine named Conpy. Conpy is easy to deploy and easy to use. Besides, Conpy runs in low level which produces meaningful reports. We make preliminary experiments to validate Conpy's capability of producing path conditions. Additionally, experiments show that the overhead of Conpy is low.

References

1. King, J.C.: Symbolic execution and program testing. J. ACM 19(7), 385–394 (1976)
2. Godefroid, P., Levin, M., Molnar, D.: Automated whitebox fuzz testing. In: NDSS, pp. 151–166 (2008)
3. Tillmann, N., de Halleux, J.: Pex-white box test generation for.NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
4. Cadar, C., Dunbar, D., Engler, D.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, pp. 209–224 (2008)
5. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. ACM Sigplan Notices 40(6), 213–223 (2005)
6. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: ESEC/FSE, pp. 263–272 (2005)
7. Fuzzgrind: An automatic fuzzing tool, `http://esec-lab.sogeti.com/dotclear/index.php?pages/Fuzzgrind`
8. Molnar, D.A., Wagner, D.: Catchconv: symbolic execution and run-time type inference for integer conversion errors. Tech. Rep. UC Berkeley EECS, 2007–23 (2007)
9. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: A platform for in-vivo multi-path analysis of software systems. Sigarch Comput. Archit. News 39(1), 265–278 (2011)
10. Xu, R.G., Godefroid, P., Majumdar, R.: Testing for buffer overflows with length abstraction. In: ISSTA, pp. 27–37 (2008)
11. Wang, T.L., Wei, T., Gu, G.F., Zou, W.: TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: S&P, pp. 497–512 (2010)
12. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A new approach to computer security via binary analysis. In: Sekar, R., Pujari, A.K. (eds.) ICISS 2008. LNCS, vol. 5352, pp. 1–25. Springer, Heidelberg (2008)
13. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: ASE, pp. 443–446 (2008)
14. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003)
15. Chen, T., Zhang, X.S., Zhu, C., Ji, X.L., Guo, S.Z., Wu, Y.: Design and implementation of a dynamic symbolic execution tool for windows executables. J. Softw-Evol. Proc. 25(12), 1249–1272 (2013)
16. Chen, T., Zhang, X.S., Xiao, X., Wu, Y., Xu, C.X., Zhao, H.T.: SEVE: Symbolic execution based vulnerability exploring system. COMPEL. 32(2), 620–637 (2013)

17. Python (programming language), http://en.wikipedia.org/wiki/Python_programming_language
18. Chen, T., Zhang, X.S., Guo, S.Z., Li, H.Y., Wu, Y.: State of the art: dynamic symbolic execution for automated test generation. Future Gener. Comp. Sy. 29(7), 1758–1773 (2013)
19. Brumley, D., Hartwig, C., Liang, Z.K., Newsome, J., Poosankam, P., Song, D., Yin, H.: Automatically identifying trigger-based behavior in malware. In: Botnet Detection, pp. 65–88 (2008)
20. Brumley, D., Hartwig, C., Kang, M.G., Liang, Z.K., Newsome, J., Poosankam, P., Song, D.: BitScope: automatically dissecting malicious binaries. Tech. Rep. CMU-CS-07-133 (2007)
21. Dive into python, everything is an object, http://www.diveintopython.net/getting_to_know_python/everything_is_an_object.html