

A Case Study of Concolic Testing Tools and Their Limitations

Xiao Qu
ABB Corporate Research
940 Main Campus Drive
Raleigh, NC, USA
xiao.qu@us.abb.com

Brian Robinson
ABB Corporate Research
940 Main Campus Drive
Raleigh, NC, USA
brian.p.robinson@us.abb.com

Abstract—Automatic testing, in particular test input generation, has become increasingly popular in the research community over the past ten years. In this paper, we conduct a survey on existing concolic testing tools, discussing their strengths and limitations, and environments in which they can be applied. We also conduct a case study to determine the prevalence of the identified limitations in six large software systems (four from open-source and two from ABB), as well as the effectiveness and scalability of the publicly available tools. The results show that pointers and native calls are the most prevalent limitations, preventing tools from generating high branch coverage test cases, and variables of float type are the least prevalent. The scalability of the publically available tools is also a limitation for industrial use, due to the large overhead of creating a test harness. Finally, we propose suggestions on how practitioners can use these tools and how researchers can improve concolic testing.

Keywords-concolic testing; automatic test generation

I. INTRODUCTION

Unit testing is a software verification and validation method in which a programmer tests if individual units of source code are fit for use. It also allows the programmers to verify that later changes, such as refactoring, do not cause existing code to break. Therefore, unit test suites are important aspects of regression testing for a software product. Although unit testing has been a best practice for years, its adoption in industry is poor. While a few companies have successfully instituted unit testing, such as Microsoft (where 79% of developers write unit tests [1]), the majority of software developed in industry have either few unit tests, outdated unit tests or low quality unit tests, due to the high cost of creating and maintaining the test suite.

Automatic test generation exists to save developers some of the manual work needed to create and maintain test suites, and to improve test effectiveness systematically. Many techniques and tools have been created for this purpose, such as unit test frameworks, test input generation, or test selection and prioritization tools. In this paper, we focus on test input generation as a way to generate effective test suites.

There are many categories of test input generation techniques, such as *random testing* [2], *symbolic execution* [3], *concolic testing* [4], *model based testing* [5], [6] and *search based testing* [7]. Model based testing is popular in research

but is often impractical since few software programs are developed with models or formal specification. Random testing is appealing but often generates a significant number of redundant test cases with low code coverage. Symbolic execution is effective in generating high coverage tests but suffer from many limitations in practise. Concolic testing needs only code to execute and it combines symbolic and random testing to overcome some shortcomings of both. In this paper, our discussion focuses on concolic testing.

Concolic testing can generate effective test cases, which has been evaluated in previous studies. But these studies were conducted on small programs, most of which were libraries or utility functions. These software programs may not contain the limitations that large industrial software does. Due to these differences, previous study results do not describe the effectiveness and scalability of concolic testing on large systems. In order to help industrial developers determine the applicability of the technique and tools for their products, evaluation on large systems is needed.

In this paper, we first identify existing concolic testing techniques and tools (Section II), identifying which languages and platforms they run on. Next, we identify and discuss the limitations of the identified concolic testing techniques and tools (Section III). We also study the prevalence of these limitations on six large open-source and industrial software systems (Section IV), as well as how these limitations can impact the effectiveness (measured in branch coverage) of the generated test suites on large programs.

The results (Section V) show that in the six systems studied, pointers and native calls are the most prevalent limitations, while variables of float type are the least prevalent. The tools can effectively generated tests suites that yield over 60% branch coverage on average. However, scalability is still a problem for large systems: though tools can generate and execute test cases quickly for each single function, the time required to setup the testing environment and test driver can be substantial. Given the large number of functions that exist in the systems studied, these techniques do not scale to the entire system yet. These results lead to further discussions in Section VI. Related work is introduced in Section VII, and finally a summary is presented in VIII.

II. CONCOLIC TESTING AND TOOLS

A software program is tested by executing *test inputs*, generating outputs and checking outputs against the *test oracles* for correctness. A complete test case is composed of test inputs and a test oracle. It is very hard to automatically generate test oracle from source code alone. In this section, we introduce concolic testing, which is a test input generation technique, and identify existing techniques and tools.

A. Concolic Testing

Concolic testing [4] is a hybrid software verification technique that interleaves concrete execution with symbolic execution. It explores executable paths in the same way as symbolic execution — unit tests are constructed to map symbolic inputs to function parameters, the techniques then collect constraints of these symbolic inputs along a set of execution paths and solves these constraints with a constraint solver — resulting in the creation of test inputs that potentially yield high path coverage. When these techniques reach difficult constraints, they use concrete values from execution to allow the algorithm to continue — this has advantages over symbolic execution, as concolic techniques can reason precisely about complex data structures, as well as simplify constraints when they exceed the capabilities of a constraint solver. In addition, since the algorithm does concrete executions, all bugs inferred by the technique are real.

DART [8] is one of the pioneer tools of concolic testing. It detects standard errors such as program crashes, assertion violations, and non-termination. SMART (Scalable DART) [9] extends DART by using a more efficient search algorithm. SAGE [10] is also built on DART. It differs from other concolic testing tools in two aspects. First, it adopts a machine-code-based approach that allows SAGE to be run on any target program regardless of its source language or build process. Second, it uses offline trace-based constraint generation, so that constraint generation in SAGE is completely deterministic.

CUTE, jCUTE [11], and CREST [12] are independent tools created in the same family: CUTE and CREST are used for C programs while jCUTE is created for Java programs. Empirical study of CREST shows its effectiveness in unit testing programs with 100-2000 lines of code [12].

EXE [13], KLEE [14] and Rwsset [15] are another family of concolic testing tools for C programs. KLEE extends EXE to address path explosion by allowing interaction with the outside environment without using entirely concrete procedure call arguments. Rwsset presents a largely complementary technique that prunes redundant paths by tracking the memory locations that are read by, and written to, the checked code in order to determine when the remainder of a particular execution is capable of exploring new behaviors.

JFuzz [16] is a tool built on JPF for concolic testing of Java programs, and Pex [17] performs concolic testing for .NET applications. In addition to generating test inputs that cover different program behaviors, Pex makes suggestions to the programmer on how to fix bugs it detects. Finally, PathCrawler [18] works on ANSI C and C++.

Tools also exist for testing web applications. Wassermann et al. [19] propose an algorithm to automatically generate test inputs. As in the standard concolic testing framework, their algorithm gathers constraints during symbolic execution. When resolving constraints over multiple types, the algorithm considers each variable instance individually.

Artzi et al.[20] apply dynamic symbolic execution to the domain of dynamic Web applications. The technique automatically generates tests to detect failures. It also minimizes the conditions on the inputs that expose failures, resulting in small but useful bug reports for detecting and fixing underlying faults. Their tool Apollo implements the technique for PHP. It generates test inputs, monitors the application for crashes, and validates that the output conforms to the HTML specification.

B. Publicly Available Tools

Tools identified in the survey, along with their supporting languages, platforms, and underlying constraint solvers are shown in Table I. Unfortunately, many of the tools are not available for public use. For instance, DART, CUTE, jCUTE, PathCrawler, and SAGE are all proprietary. The tools in *bold* are publicly available.

As mentioned in the beginning of this section, test input is only one part of a complete test case. The test oracle, as the other part, is also critical. Though this paper does not discuss the test oracle, it is included in the table to illustrate how these tools work as a whole piece. Oracles are classified as *Uncaught Exceptions* (UE), *Operational Model* (OM), or *Unknown* (NA). Tools classified as UE regard a test case as potentially faulty if it throws an uncaught exception or violates manually inserted assertions (i.e., crashes). Tools classified as OM develop an operational model from sample tests or executions, and the model can be complemented with manually written specifications. By inferring the operational model, properties such as objects invariants or method pre/post conditions are checked, which can be manually written in the code or transformed from formal specifications. If these properties are violated, the test case may reveal potential faults. Finally, tools classified as NA do not specify how their testing oracles operate in the publications.

III. LIMITATIONS

Theoretically, given a powerful constraint solver, the concolic testing tools are expected to generate test cases that can cover all reachable statements or branches. However, this constraint solver does not exist, leading to many limitations

Table I
CONCOLIC TESTING TOOLS
(**BOLD** TOOLS ARE PUBLICLY AVAILABLE)

Tool	Language	Platform	Oracle	Constraint Solver
DART [8]	C	NA	UE	lp_solver
SMART [9]	C	Linux	UE	lp_solver
CUTE [11]	C	Linux	UE	lp_solver
jCUTE [11]	Java	Linux	UE	
CREST [12]	C	Linux	UE	Yices
EXE [13]	C	Linux	UE	STP
KLEE [14]	C	Linux	UE	STP
Rwset [15]	C	Linux	UE	STP
JFuzz [16]	Java	Linux	UE	built on JPF
PathCrawler [18]	C	NA	NA	NA
Pex [17]	.NET	Windows	UE	Z3
SAGE [10]	machine code	Windows	UE	Disolver
NA [19]	PHP	NA	NA	NA
Apollo [20]	PHP	NA	NA	NA

shared by various tools. Moreover, specific tools have their own limitations due to different implementations. In this section, we discuss these limitations. Many of the limitations are presented in publications or/and on-line documents, in addition, we contact the authors for completeness.

A. Assumptions (Out-of-scope Limitations)

One limitation of concolic testing involves analyzing multi-threaded code. While some research has started to address concurrency and non-deterministic execution, most concolic testing tools assume a program is single-threaded and deterministic. This paper holds the same assumption.

Some publications regard structures as a limitation. However practically, even if a whole structure or object can not be declared as symbolic, it can be treated as multiple primitive data types — though it may require extra time to specify symbolic inputs and necessary constraints. This paper does not consider this a limitation.

B. Limitations of Concolic Testing

Similar to symbolic execution, concolic testing aims to generate test inputs that result in high path coverage. Its capability is highly dependent on the effectiveness of its underlying constraint solver: most constraint solvers do not support float or double data types, and some solvers are unable to handle non-linear arithmetic constraints. When the constraint is beyond the ability of the solver, symbolic execution is unable to proceed further along that path and switches to explore different paths. Different from this, concolic testing can overcome this by substituting symbolic inputs with concrete values. While this has the potential to allow continued exploration of a stalled path, the technique devolves to a form of random testing and loses the advantages of exploring new behaviors. From this perspective, we still consider them as limitations in concolic testing, which are defined as **float/double data type variables** and **non-linear arithmetic operations** (they include multiplication,

division, and modular; shifting and masking will be separately treated as bitwise operations).

There are other typical limitations of concolic testing that are inherited from symbolic execution. For example, it cannot handle calls to code that are invisible [3]. We define this limitation as **native calls**. These native calls involve standard library calls or employment of third party components whose source code is unavailable. But for Java and C#, standard library calls may not be a limitation, as most techniques work at the bytecode level.

Some limitations are specific to tools. **Pointer** is one instance. DART only handles integer constraints and devolves to random testing when pointers are encountered [13]. SAGE does not currently reason about symbolic pointer dereferences [10]. In contrast, KLEE is able to handle pointers, although marking a pointer by itself as symbolic can lead to a rapid state explosion. Again for Java and C#, pointers are not a problem since it does not exist.

Other tool-specific limitations include **symbolic offsets** and **function pointers**. Specifically, CUTE does not handle symbolic offsets but EXE does [13]. Neither EXE nor CREST currently performs any static reasoning about calls through function pointers [12]. Some tools cannot deal with **bitwise operations**, such as JFuzz [16]. CREST is a little special in that it handles shifting but not masking.

We illustrate a few limitations in two sample functions shown in Figure 1. Statement 1 contains a limitation of pointer because y is defined as a pointer and used in statement 1, a conditioning statement (i.e., *if* statement); statement 3 contains limitations of native call, pointer, and non-linear arithmetic operation; statements 5, 7 and 8 involve variables of float type.

To summarize, typical limitations shared by concolic testing tools include: (1) variables of float/double type; (2) pointers; (3) native calls; (4) non-linear arithmetic operations; (5) bitwise operations; (6) array offsets and (7) function pointers. Table II contains a list of the limitations, as well as identifying whether each tool has the limitation or not. A “N” means the limitation does not exist for the particular tool, i.e., the tool can handle the corresponding limitation. For example, CUTE can handle pointers. A “P” means the tool can support part of the limitation. For example, CREST can handle shifting but not masking for bitwise operation. A “Y” means the tool cannot handle the limitation at all, resulting in decreased effectiveness when generating tests for functions that contain that limitation.

C. Issue of Scalability

In addition to the above limitations, concolic testing may also suffer from scalability concerns. This section discusses this issue, followed by a case study in Section IV-D.

Previously, most evaluations of symbolic and concolic testing tools focus on libraries or utility functions. These programs have the advantage of being able to be tested

Table II
LIMITATION OF CONCOLIC TESTING TOOLS

Tool Name	Limitations						
	float/double	pointer	native call	non-linear arithmetic op.	bitwise op.	offset	function pointer
DART	Y	Y	Y	NA	NA	Y	Y
SMART	Y	Y	Y	NA	NA	Y	Y
CUTE	Y	N	Y	NA	NA	Y	Y
jCUTE	Y	-	Y	NA	NA	Y	Y
CREST	Y	Y	Y	P	P	Y	Y
EXE	Y	N	Y	N	N	N	Y
KLEE	Y	N	P	N	N	N	NA
Rwset	Y	N	Y	N	N	N	NA
Jfuzz	Y	-	Y	Y	Y	NA	NA
Pex	Y	-	Y	NA	NA	NA	NA
SAGE	NA	Y	N	NA	NA	NA	NA
PathCrawler	NA	NA	Y	NA	NA	NA	NA

```

typedef char* STR;
typedef float REAL;

REAL a;
STR b;

void function1(int x, char *y)
{
1.   if(y != NULL)
2.       b = y;
3.   if((strlen(b) > x) && (x % 4 == 0))
4.       // ....
}

void function2(float x)
{
5.   if(x < 0) // ...
6.   else
7.       if(sqrt(a/3.14) == x) //...
8.       if(a == 2*x*3.14) // ...
}

```

Figure 1. Sample Program

as a single unit, because these programs are small and all their methods cohesively implement single functionalities. A small set of inputs to an *API* function controls a large portion of the code in the program. However, the situation is different in large industrial systems. These systems are much bigger and they are usually composed of multiple components that represent multiple independent functionalities. It is not possible to test them all together as a single unit. In order to run tools on these systems more effectively, we must divide the systems into multiple testable units. This introduces the first cost, the time to partition system (tp).

After partitioning the system into units, the cost needed to test each unit of a program can be broken up into three parts: time to setup environment and tool (ts_1), time to create test driver and specify symbolic inputs and constraints (ts_2), and the time to generate inputs while executing test cases (tt). Time ts_1 varies, as each tool has different requirements. For example, KLEE requires the source to be compiled into LLVM bytecode instead of a linked executable, CREST runs on normal executables, and JFuzz runs on class files. Time ts_2 also varies. Simple units with limited constraints will not take much time, while units with complex constraints

and a large number of symbolic inputs can take significant time to specify.

By considering all the costs discussed above, the total cost is calculated as $tp + \sum_{i=1}^N (ts_i + tt_i)$, where N is the number of units, and ts , the setup time, is the sum of ts_1 and ts_2 that are calculated together in later study. To minimize the cost, we can either reduce tp or ts and N , while tt is out of our control since it reflects the capability of the tool itself.

In order to lower tp , a unit can be simply defined as a single function. However, in large systems, this type of partition may result in a huge number of units and may also increase the setup time, because few functions in large programs are as independent as utility functions. In other words, functions in one component usually couple with each other in a high degree — they call functions in other source files, sometimes in other components. This requires the tester to identify these dependencies and make sure they are included in the test build correctly. The time required to perform this can be substantial.

A good way to reduce both N and ts is to divide the system into high-cohesion and low-coupling units that are appropriate to apply concolic testing tools on, while not sacrificing the effectiveness (i.e., code coverage). This involves a large effort in partitioning, leading to a much higher tp .

Therefore, to test a large system using concolic testing involves a tradeoff between cost of partitioning and setting up units. To date, not many strategies exist to partition large systems. Chakrabarti et al. [21] present an approach by identifying the data and control inter-dependencies between components, but to collect the data and control inter-dependencies for large systems is another difficulty. We leave this as future work (Section VIII).

In later study, we define each single function as a unit, and investigate the scalability under this circumstance.

IV. STUDY OF LIMITATIONS AND EFFECTIVENESS

In this section, we conduct a study of the prevalence of limitations identified in Section III-B, on six open-source and industrial software systems. We also examine the effectiveness and scalability of the publically available

tools when applied to a large industrial system. This study aims to answer the following research questions.

- **RQ1:** How prevalent are the typical limitations of concolic testing in large software systems?
- **RQ2:** How effective are concolic testing tools when applied to industrial systems?
- **RQ3:** Is scalability a problem of concolic testing tools when applied to large systems?

A. Object of Analysis

The subject programs in this study are large software systems selected from both industry and open-source that are written in C and Java. These languages are selected because they are the most frequently used languages in practice, and there exist publicly available concolic testing tools for them. We select four open-source systems from the top 25 most evaluated open-source programs [22]: *gcc* (version 4.3.0) and *linux* (version 2.6.25) for C, *Eclipse* (version 3.4.2) and *Jboss* (version 4.2.2GA) for Java. In addition, two industrial systems developed at ABB are studied: a large C program we call ABB1 (a realtime embedded system) and a large Java program we call ABB2 (an industrial GUI system). Metrics on the executable lines of code (LOC) and number of functions (NOF) for each program are shown in Table III, calculated by the *Understand* commercial metrics tool [23].

For RQ2 and RQ3, two publicly available concolic testing tools for C programs, CREST (revision 131) and KLEE (revision 115315 built with llvm 2.7, incorporating *uClibc*) are selected to test ABB1¹. These tools have been previously evaluated on open-source systems. For example, CREST [12] was shown to generate test cases that yield 80%, 60% and 19% branch coverage for *replace*, *grep* and *vim*. KLEE [14] was evaluated on 89 GNU COREUTILS utilities, yielding average 90% statement coverage.

Table III
BASIC INFORMATION OF SYSTEMS

	gcc	linux	ABB1	Jboss	Eclipse	ABB2
LOC	1,986,052	4,454,584	3,911,190	633,156	2,455,821	142,688
NOF	92,936	171,454	134,074	60,753	234,002	12,189

B. Prevalence of Limitations

We measure the prevalence of limitations in three different ways: the **percentage** of functions in a system that contain one specific limitation, the **density** of one specific limitation in a system — average occurrence of this limitation in each function, and the **blocking factors** of one specific limitation — in each function, a blocking factor is the percentage of *branching points* (e.g., if-else, for) at which the concolic testing is prevented by this limitation from further exploring, it ranges from 0 to 1. We develop this metric because a

¹JFuzz is the only Java tool available to public but it has no active maintainer from which we could get help regarding the use of the tool.

key benefit of concolic testing is to explore new paths at branching points by negating and solving constraints, we expect it to provide us with more accurate insights.

The limitations are identified by SrcML [24], a multi-language fact extractor. After srcML translates each program into its XML-like markup language, queries are created to find occurrences of the limitations (we do not show details on this due to space limit). Among the seven identified limitations discussed in Section III-B, we address five in this study: floats, pointers, native calls, non-linear arithmetic operations, and bitwise operations (annotated as limitation 1 to 5). The other two are not included due to the difficulty of designing queries to identify them. Metrics captured from the results of the queries are shown as follows:

- total Number Of Functions (NOF)
- Number Of Functions that contain a specific Limitation i (NOLF_i , $i=1$ to 5)
- Number Of Occurrences of Limitation i in function j (NOOL_{ji})
- Number Of Branch points that encounter specific Limitation i in function j (NOBL_{ji})
- Number Of Branch points in each function j (NOB_j)

With the above data, the prevalence of limitation i is measured by following metrics as defined before, where j represents each function under test.

- 1) $\text{percentage}_i = \text{NOLF}_i \div \text{NOF}$
- 2) $\text{density}_i = \sum_j \text{NOOL}_{ji} \div \text{NOLF}_i$
- 3) $\text{blocking factor}_{ji} = \text{NOBL}_{ji} \div \text{NOB}_j$

The calculation process is illustrated below, taking the program in Figure 1 as an example. There are two functions in total (i.e., $\text{NOF}=2$). In function 1, there are two branching points at statement 1 and 3 ($\text{NOB}_1=2$). At statement 1, one pointer limitation (i.e., y) is encountered; at statement 3, one native call (i.e., *strlen*), one pointer (i.e., b) and one non-linear arithmetic (i.e., $\%$) are encountered. In summary, function 1 contains two pointers ($\text{NOOL}_{12}=2$), one native call ($\text{NOOL}_{13}=1$), and one non-linear arithmetic operation ($\text{NOOL}_{14}=1$). There are two branches that contain pointer ($\text{NOBL}_{12}=2$), one branch that contains native call ($\text{NOBL}_{13}=1$), and non-linear arithmetic operations ($\text{NOBL}_{14}=1$). The same process is repeated to function 2.

Considering the whole program composed of both functions, there is one function that contains floats ($\text{NOLF}_1=1$), two functions that contain native calls ($\text{NOLF}_3=2$) and so on. To the end, the prevalence of float (i.e., limitation 1) is measured by the metrics: 1) $\text{percentage}_1 = \text{NOLF}_1 \div \text{NOF} = 50\%$, 2) $\text{density}_1 = (\text{NOOL}_{11} + \text{NOOL}_{21}) \div \text{NOLF}_1 = 3$, and 3) $\text{blocking factor}_{11}$ and $\text{blocking factor}_{21}$ are equal to 0 and 1.

Particularly, for the language dependent limitations (such as pointer), our analyses are restricted to programs written in languages that suffer from the limitations.

C. Effectiveness of Tools

As explained in Section III-C, this study treats each individual function as a unit, which means all functions in system ABB1 are candidate testing objects. ABB1 contains a large number of functions, and each function differs in size and complexity, which may impact the testing effectiveness and test time. Due to this impact, we do not randomly select functions from the whole system but select from different categories based on levels of complexity. *Cyclomatic complexity* is the standard metric that we use to measure the complexity, it is calculated by Understand [23] as $Edges - Nodes + Connected Components$. We break the functions into three categories based on their complexities (LOW for a complexity 1-4, MID for 5-10, and HIGH for greater than 10). ABB1 has 61.5% of its functions with a complexity between 1 and 4, 17% of its functions' complexities are between 5 and 10, while 11.5% of its functions' complexities are greater than 10.

We randomly select 20 functions from each category, and use CREST and KLEE to create tests for each function. The size of selected functions range from 6 to 214 lines of code and the complexity ranges from 1 to 59. The total lines of code of all 60 functions are 3081.

We use branch coverage to measure the effectiveness of the tools. CREST automatically calculates the coverage over all reachable branches by using an integrated functionality. KLEE also calculates coverage but it is not accurate, due to: first, it works at the instruction level instead of the source code level; second, it also counts internal instructions that do not belong to our testing objectives. Therefore, we use *gcov* [25] to calculate branch coverage for KLEE.

In order to investigate the best capability of both tools, when encounter with build or execution failures or unexpected low coverage in using the tools, we try many times to modify or add symbolic inputs and constraints, until the problem is resolved or we are unable to find a solution. This potentially increases the setup time. Moreover, in order to achieve consistent results, we create test drivers with the same symbolic instrumentations for CREST and KLEE. For example, we initialize symbolic arrays with the same values and sizes.

D. Scalability of Tools

Even if the tools are effective to generate high coverage test cases, it is difficult to be applied in practice if it costs too much. This section studies this scalability issue.

As discussed in Section III-C, the time to test a large system can be broken up into: time to partition system (tp , once per system), time to setup tool and environment plus time to define symbolic inputs and constraints (ts , each test unit), and time to generate and execute tests (tt , each test unit). This study defines each function as a unit ($tp = 0$), the total cost is decided by ts and tt .

Test time is first studied as the indicator of the tools' capability. Since this time may be impacted by the complexity of functions, it is calculated separately for functions whose complexities are at different levels. For each function i whose complexity is at level j ($j = 1$ to 3, representing LOW, MID, and HIGH), we collect its test time tt_{ji} . We then estimate the time of testing functions whose complexities are at the same level, by using the average time of selected 20 functions: $(\sum_{i=1}^{20} tt_{ji} \div 20) \times (N \times p_j)$, where N is the total number of functions and p_j is the percentage of functions whose complexities are at the same level j (for example, $p_1 = 61.5\%$). By summing up the time of functions with complexities at three levels, the test time of the whole system can be estimated by Formula 1.

$$tt = \sum_{j=1}^3 [(\sum_{i=1}^{20} tt_{ji} \div 20) \times (N \times p_j)] \quad (1)$$

Though setup time is not directly reflect the tools' capability, it is a significant cost when we consider about scalability. As discussed in Section III-C, setup time may also be impacted by the complexity of functions. Its calculation is similar to test time, by replacing tt with ts in Formula 1. To the end, the overall cost is $T = tt + ts$.

E. Threats to Validity

It is possible that our fact extractor missed some occurrences of limitations in the systems we studied, which would limit our recall. To reduce these occurrences, we identified alias names for many data types, such as "float", "double" and "char *" which are defined by "typedef" or "#define", and counted them in the occurrence of these limitations. For example, the "STR" and "REAL" are pointer and float alias in program in Figure 1. However, some of the limitations may be more difficult to catch, such as native calls that are embedded in macros or function pointers.

While we select programs from both open-source and industrial software to study, they may not represent all of the possible domains of software programs. Therefore, the study results on these systems may not be generalizable.

The coverage calculated by CREST and KLEE is not the net coverage of the unit under test, as the tools also count the code in test drivers. Moreover, KLEE also counts unreachable code in the denominator of calculating coverage, which makes the results lower than the true values.

V. RESULTS

In this section, we present the results of the two studies, regarding to the prevalence of typical limitations of concolic testing in large systems, the effectiveness of the tools when these limitations are present, and the scalability of concolic testing tools.

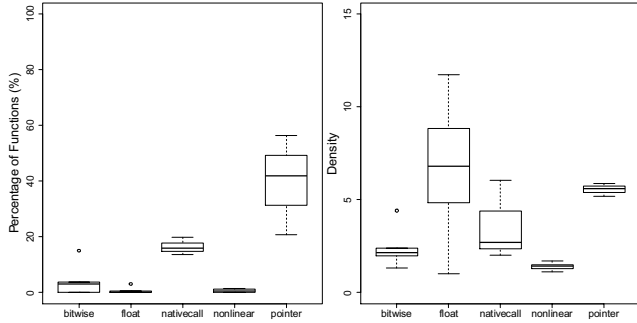


Figure 2. Percent. and Density of Limitations

A. RQ1: Prevalence of Limitations

The results presented in this section include aggregate data from all six subject programs. The percentage of functions that contain limitations, as well as the densities of these limitations, are illustrated as two sets of boxplots in Figure 2. The results in the left set show that pointers and native calls are the most prevalent limitations (with 50% and 20% of the functions containing them, respectively) while float is the least prevalent. The results in the right set show that floats have the highest density (at more than six occurrences per function). Pointers have the second highest density with over five occurrences per function, followed by native calls with three occurrences per function.

The lowest occurrence rate but the highest density of float implies that the usages of floats are concentrated in a small number of functions. The most prevalent limitations — pointers and native calls — are also quite dense. For example, over 50% of the functions contain pointers and, on average, each of those functions contains six pointers. Similarly, 20% of the functions contain native calls, each containing three calls an average. From the perspective of improving the overall effectiveness of concolic testing, enhanced support for pointers and native calls will lead to the largest improvements.

The results of the blocking factor metric are shown as boxplots in Figure 3. On average, pointers prevent 60% of a function’s branches from being explored by concolic testing, and native calls prevent 50% of them. Their values are higher than the values of other limitations, indicating their significances in preventing concolic testing from exploring more paths and generating higher code coverage tests. This is consistent with the results of percentage and density.

B. RQ2: Effectiveness of Tools

We have encountered with some unexpected problems in this study. For example, with initial definitions of symbolic inputs and constraints, KLEE failed to generate test cases for three functions. The problem was solved by modifying the test driver. CREST ran into runtime errors for seven functions, but we were unable to resolve the problems. In the results illustrated below, we regard their coverage as zero.

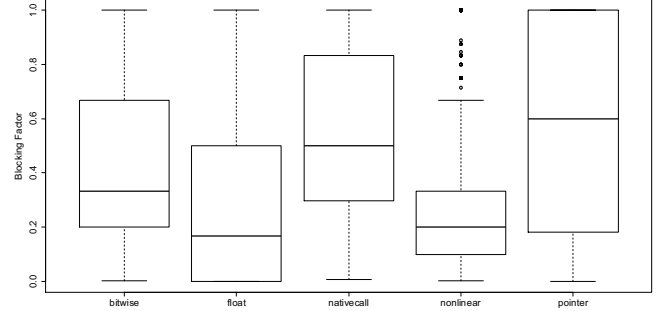


Figure 3. Blocking Factor of Limitations

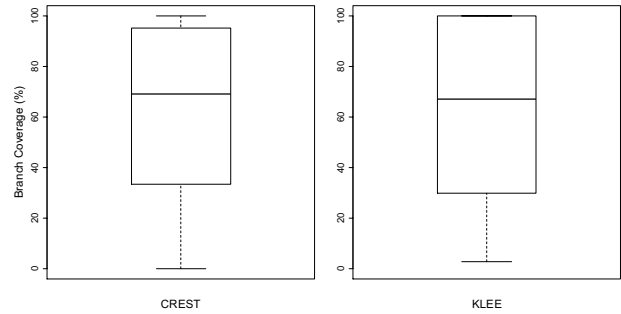


Figure 4. Branch Coverage

The coverage results of all 60 functions are shown in Figure 4. Excluding the failures, CREST and KLEE produce test cases that achieve an average of over 60% branch coverage². Both tools generate test suites that yield full coverage for functions across all levels of complexity (raw data omitted for space reasons). They also generate test suites that yield low coverage, led by the limitations that exist in the systems. For example, most of the selected functions contain pointers, and half of them call external library functions. A few functions also contain non-linear or bitwise operations, as well as variables of float type. More findings are discussed in Section VI.

C. RQ3: Scalability of Tools

First, we study the test time (tt). Figure 5 shows the test time used by KLEE and CREST to generate and execute test cases for functions grouped by different levels of complexity. Generally, the test time for each function is quite small, as most need under 30 seconds to run, with a few outliers taking around one minute (complete data omitted due to space limit).

Though the figures shows a positive correlation between complexity and test time, the fact is that for some high complexity functions, it takes only a few seconds to generate test cases. These deviations are due to two factors. First, limitations prevent the tools from exploring more paths,

²As mentioned before, this is not net coverage of function under test. In addition, the results of CREST and KLEE are not comparable since they calculate it using different tools and methods.

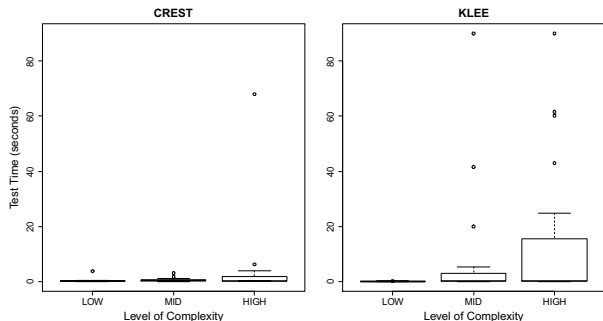


Figure 5. Test Time (tt) of Different Complexities

resulting in a short execution. Second, it is related to the number of symbolic inputs and constraints. A complex function with not many symbolic inputs does not bother the constraint solver frequently (the constraint solving is very costly), as many code paths are executed concretely. We conduct a ANOVA test to quantitatively study the correlation between complexity and test time, the result shows no significant correlation. Therefore, the test time can be calculated without considering different levels of complexity, by modifying the Formula 1 to Formula 2. There are 134,074 functions in ABB1, as a result, the total test time is 70 hours and 180 hours by CREST and KLEE.

$$tt = \left(\sum_{i=1}^{60} tt_i \div 60 \right) \times N \quad (2)$$

While the test time ended up being quite small, the setup time is substantial. Regardless of the complexity of the function, it takes an average of 10 minutes to set up the environment (ts_1) for each function, and then it takes 5 minutes for CREST and 4 minutes for KLEE to specify the symbolic inputs and constraints. There is no significant difference of setup time between different complexities of functions, so the total setup time can also be calculated by Formula 2, with tt replaced with ts . Given the number of functions, 134,074, the total setup time is 33,518 and 31,283 hours for CREST and KLEE.

Adding test time and setup time together, the total cost to test the whole system of ABB1 will be 33,589 hours by CREST and 31,464 hours by KLEE. The setup time takes up more than 99% of the cost. Even though we expected this cost to be big, as discussed in Section III-C, it is much larger than expected. This result indicates that using concolic testing tools to test every single function is very inefficient in a large system, primarily due to the setup cost and the number of functions. We need a better strategy to use the tools, as discussed in Section VI.

D. Additional Observations

Results of the RQ1 show that native calls are the second most prevalent limitation. These calls are usually system

or library calls. Replacing these calls with stub functions can improve the coverage of the produced test cases. To study this further, we calculated the occurrence rates of six different GNU C standard library calls. The results (Table IV) show that, on average, string manipulation functions and the *sizeof* function are the most frequently called. IO operations are the next most frequently called. Finally, math functions are also frequently called in general purpose applications. Specifically in ABB1, math functions are called more than 1560 times.

Table IV
PREVALENCE OF FREQUENT C LIBRARY CALLS

Objects	str	ctype	stdio	stdlib	math	sizeof
gcc	984	10	232	1	0	358
linux	1	216	66	105	0	5698
ABB1	5992	640	1545	207	1561	2843

VI. DISCUSSIONS AND SUGGESTIONS

In this section, we discuss some results and propose suggestions to improve concolic testing.

The results of RQ1 show that pointer and native calls are the most prevalent limitations in the systems studied. The results of RQ2 verify their negative impacts on coverage. Mitigating these limitations may largely improve the effectiveness of the tools. Some limitations have been addressed by works introduced in Section VII-B. Some limitations are tool specific, such as non-linear arithmetic operations and offsets, which can be overcome by updating the SMT solver (Table I, last column) or changing the tool implementation.

There are some unexpected results of coverage, we discuss them below. KLEE is expected to produce high coverage for most functions, since it handles both pointers and native calls by incorporating *uClibc*, a C library optimized for embedded systems. But the results show that about 30% functions are covered less than 50% branches. This is mostly caused by native calls outside *uClibc* – after all *uClibc* is smaller than the *GNU C Library* that the real programs use. Mathematic methods such as *sqrt*, frequent standard library calls such as *sizeof*, and type casting are still not addressed. A more complete simulation, especially of the frequently called ones as counted in Table IV, will be more helpful. Alternatively, a string constraint solver may be used.

The results of RQ3 indicate that to apply concolic testing tools to large systems, two tradeoffs are involved. First, there is a tradeoff between partition time and tool setup time (i.e., tp and $ts_1 \times N$). In our study, the setup time is so substantial because we regard each function as a unit, and at the function level, units are high coupling with each other. This requires us to identify these dependencies and make sure they are included in the test build correctly. The time required to perform this setup is non-trivial. Moreover, this leaves us with the largest number of units. This indicates that we need a higher granularity definition of units, but how fine the

partition is still needs careful consideration — the finer the definition, the smaller the $ts_1 \times N$, but the higher the tp .

There is another tradeoff between the coverage (effectiveness) and cost to create test driver (i.e., ts_2). In our study, we have spent a lot time to try different settings of symbolic inputs and constraints, to maximize the effectiveness of the tools. Practitioners need to balance between these given different requirements (e.g., coverage) and cost limit (e.g., time).

Moreover, our study adds extra overheads to the setup time (but not included in the calculation of ts): our system is developed in Windows but both CREST and KLEE work in Linux. We have to transplant every function from Windows to Linux and make them build on Linux. Each function costs us more than 30 minutes beyond the regular setup time which we count in final calculations. To spend time on this transplant does not contribute to the goal of our study, so we select only 20 functions from each function group of different complexities. However, this suggests that tools on different platforms are needed.

VII. RELATED WORK

A. Surveys on Concolic Testing

Păsăreanu et al. [3] conduct a survey of new research trends in symbolic execution, with particular emphasis on applications to test generation and program analysis. This paper focuses on the future research directions of symbolic testing while we focus on limitations in practical use.

Sen [4] introduces the underlying concept of concolic testing and existing tools. This paper does not talk about limitations or practical applications.

B. Study of Limitations

Many techniques have been proposed to address pointers in concolic testing. Visvanathan et al. [26] present a technique to generate test data for functions with pointer inputs, by solving a set of pointer constraints and generating the shape of the input data structure. Instead of eagerly enumerating all possible heap shapes, Vanoverberghe et al. [27] automatically generate disjointness constraints for memory, to help the constraint solver generate relevant heap shapes for testing. This approach is implemented in `Pex`. Yogi [28] implements an algorithm enhanced to handle pointers and procedures.

There is also research addressing string manipulations, an essential part of the native library calls. Shannon et al. [29] demonstrate an approach to symbolically represent strings, by abstracting away the implementation details of strings using finite-state automata. Bjorner et al. [30] abstract the string constraints and pass it to an SMT solver and the solver either returns unsatisfiable or a fixed-length string. HAMPI [31] is a solver for string constraints over fixed-size string variables. It is integrated in JPF.

The limitation involving floats has also been studied. Sy et al. [32] present a novel approach for automated test data generation of imperative programs containing float variables. Lakhota et al. [33] introduce an approach that uses a search-based software testing technique to handle floating point computations. Godefroid et al. [34] propose an approach to address floating-point (FP) by combining static analysis of FP instructions with dynamic analysis of non-FP instructions.

As for the scalability, Chakrabarti et al. [21] present an approach to partition large software systems into appropriate units that can be tested in isolation.

Lakhota et al. [35] study a concolic tool, CUTE, and a search based tool, AUSTIN on five applications. They provide an assessment of both tools' limitations but do not study the frequency, nor do they summarize the limitations shared by other tools.

VIII. SUMMARY AND FUTURE WORK

In this paper, we conduct an up-to-date survey on concolic testing technique and tools, providing practitioners with a guide to select tools for different software products. We illustrate and discuss their limitations, and study the prevalence of these limitations on six large open-source and industrial software systems. We also study the effectiveness and scalability of these tools.

The results show that pointers and native calls are the most prevalent limitations, while float variables are the least prevalent. In order to improve the effectiveness of the tools in the presence of these limitations, a string constraint solver may be used, or stubbing of frequently used C library functions may be utilized. Moreover, current tools exist mostly for C on Linux. Developers writing code in Windows, or in C++, have limited choices.

CREST and KLEE can produce test cases with average over 60% branch coverage in only a few seconds, for single functions. But given the substantial overhead of setup time and the huge number of functions in systems, it is unable to test the whole system within a few days. We do not discourage practitioners to use the tools, but we need to use them in a better manner such as defining a unit of a different granularity. Finally, higher coverage may require more time to specify symbolic inputs and their constraints, practitioners need to make balance between these given different requirements and cost.

REFERENCES

- [1] G. D. Venolia, R. DeLine, and T. LaToza, "Software development at microsoft observed," Microsoft Research, Tech. Rep. MSR-TR-2005-140, 2005.
- [2] J. H. Andrews, S. Haldar, Y. Lei, F. Chun, H. Li, and N. B, "Randomized unit testing: Tool support and best practices," Department of Computer Science, University of Western Ontario, Tech. Rep., 2006.

- [3] C. S. Păsăreanu reanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 4, pp. 339–353, 2009.
- [4] K. Sen, "Concolic testing," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 571–572.
- [5] J. J. Gutierrez, M. J. Escalona, M. Mejias, and J. Torres, "Generation of test cases from functional requirements. a survey," Workshop on System Testing and Validation, 2006.
- [6] M. Prasanna, S. Sivanandam, R. Venkatesan, and R. Sundarajan, "A survey on automatic test case generation," *Academic Open Internet Journal*, vol. 15, 2005.
- [7] M. Harman, "Automated test data generation using search based software engineering," in *Proceedings of the Second International Workshop on Automation of Software Test*, 2007, pp. 2–.
- [8] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 213–223.
- [9] P. Godefroid, "Compositional dynamic test generation," in *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007, pp. 47–54.
- [10] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of Network and Distributed Systems Security*, 2008.
- [11] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," *Computer Aided Verification*, pp. 419–423, 2006.
- [12] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-123, 2008.
- [13] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: automatically generating inputs of death," in *Proceedings of the 13th ACM conference on Computer and communications security*, 2006, pp. 322–335.
- [14] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *USENIX Symposium on Operating System Design and Implementation*.
- [15] P. Boonstoppel, C. Cadar, and D. Engler, "Rwset: Attacking path explosion in constraint-based test generation," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.
- [16] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, "jfuzz: A concolic whitebox fuzzer for java," in *Proceedings of the First NASA Formal Methods Symposium*, 2009, pp. 121–125.
- [17] N. Tillmann and J. de Halleux, "Pex-white box test generation for .net," in *Proceedings of the Second International Conference on Tests and Proofs*, 2008, pp. 134–153.
- [18] C. Marcondes, M. Sanadidi, M. Gerla, R. S. Schwartz, R. O. Santos, and M. Martinello, "Pathcrawler: Automatic harvesting web infra-structure," in *Proceedings of the IEEE Network Operations and Management Symposium*, 2008, pp. 339–346.
- [19] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proceedings of the 2008 international symposium on Software Testing and Analysis*, 2008, pp. 249–260.
- [20] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in dynamic web applications," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, 2008, pp. 261–272.
- [21] A. Chakrabarti and P. Godefroid, "Software partitioning for effective automated unit testing," in *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, 2006, pp. 262–271.
- [22] B. Robinson and P. Francis, "Improving industrial adoption of software engineering research: A comparison of open and closed source software," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2010, pp. 21:1–21:10.
- [23] S. T. Inc., <http://www.scitools.com/documents/metrics.php>.
- [24] M. Collard, "Addressing source code using srcml," in *IEEE International Workshop on Program Comprehension Working Session: Textual Views of Source Code to Support Comprehension*, 2005, pp. 109 – 112.
- [25] F. S. Foundation, "gcov."
- [26] S. Visvanathan and N. Gupta, "Generating test data for functions with pointer inputs," in *Proceedings of the 17th IEEE International Conference On Automated Software Engineering*, 2002, p. 149.
- [27] D. Vanoverberghe, N. Tillmann, and F. Piessens, "Test input generation for programs with pointers," in *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2009, pp. 277–291.
- [28] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur, "The yogi project: Software property checking via static analysis and testing," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2009, pp. 178–181.
- [29] D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid, "Abstracting symbolic execution with string analysis," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, 2007, pp. 13–22.
- [30] N. Björner, N. Tillmann, and A. Voronkov, "Path feasibility analysis for string-manipulating programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2009.
- [31] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "HAMPI: A solver for string constraints," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, 2009, pp. 105–116.
- [32] N. T. Sy and Y. Deville, "Automatic test data generation for programs with integer and float variables," in *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, 2001, p. 13.
- [33] K. Lakhotia, N. Tillmann, M. Harman, and J. De Halleux, "Flopsy: search-based floating point constraint solving for symbolic execution," in *Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems*.
- [34] P. Godefroid and J. Kinder, "Proving memory safety of floating-point computations by combining static and dynamic program analysis," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, 2010, pp. 1–11.
- [35] K. Lakhotia, P. McMinn, and M. Harman, "An empirical investigation into branch coverage for c programs using CUTE and AUSTIN," *Journal of Systems and Software*, vol. 83, pp. 2379–2391, 2010.