Concolic Test Generation and the Cloud: Deployment and Verification Perspectives

Nikolai Kosmatov
CEA, LIST, Software Safety Laboratory, PC 174
91191 Gif-sur-Yvette France
Nikolai.Kosmatov@cea.fr

Abstract

Software testing in the cloud can reduce the need for hardware and software resources and offer a flexible and efficient alternative to the traditional software testing process. A major obstacle to the wider use of testing in the cloud is related to security issues. This chapter focuses on test generation techniques that combine concrete and symbolic execution of the program under test. Their deployment in the cloud leads to complex technical and security issues that do not occur for other testing methods. We describe our recent online deployment of such a technique implemented by the PathCrawler test generation tool for C programs, where we faced, studied and solved many of these issues.

Mixed concrete/symbolic testing techniques not only constitute a challenging target for deployment in the cloud, but they also provide a promising way to improve the reliability of cloud environments. We argue that these techniques can be efficiently used to help to create trustworthy cloud environments.

Introduction

Testing is nowadays the primary way to improve the reliability of software. Software testing accounts up to 50% of the total cost of software development. Automatic testing tools provide an efficient alternative to manual testing and reduce the cost of software testing. However, automatic testing requires considerable investments: purchase and installation of testing tools, additional computing resources to run these tools, employing or training competent validation engineers to maintain and operate them, etc. These resources are necessary only during the testing steps of software development, and their cost for the company outside this period can be avoided by sharing them between several projects and with other companies.

The paradigm of cloud computing brings obvious benefits for the software testing process. The deployment of software testing services in the cloud makes them easily available for different companies and projects and allows their on-demand usage. The companies do not

have to purchase and maintain powerful servers and testing tools all the time, but use them just when it is required.

On the other hand, for the providers of testing tools, this approach makes it easier to update and to support the tools and to provide flexible on-demand solutions to the clients. Various testing tasks, taking from several seconds up to several weeks, can be optimally scheduled in the cloud. Thus a testing service can be offered to a larger number of companies and becomes appropriate for testing software of almost any size.

Before testing in the cloud becomes widely accepted and used in industry, various technical, security and privacy protection issues must be resolved. In this chapter, we focus on test generation techniques combining concrete and symbolic execution, also known as *concolic testing*. Concolic testing is an advanced technique of structural unit testing, that is one of the most suitable kinds of testing for the cloud (Parveen & Tilley, 2010). We address two facets of concolic testing in the cloud: migrating concolic testing to the cloud and usefulness of concolic testing for the cloud. The deployment of concolic testing in the cloud raises particularly challenging technical and security problems that do not necessarily appear in other testing methods. Relevant to any version of concolic testing, the security and efficiency concerns become even more critical for a publicly available testing service in the cloud. While for a local deployment used by a restricted number of people, an intentionally malicious software is very unlikely to be submitted to the tool, a publicly available testing service runs a much greater risk. We have recently implemented and deployed an online version for such a technique where we faced, studied and solved many of these problems. We show how a concolic testing tool can be decomposed into safe and unsafe parts in order to preserve the efficiency of the method, and how the unsafe part can be secured. On the other hand, concolic testing provides an excellent means for improving the reliability of the cloud itself. We present the most recent results on verification of operating systems and cloud hypervisors and show the role of various concolic testing approaches for creating more reliable cloud environments.

We start by providing some background on testing in the cloud and migrating software testing to the cloud. Next, we give an overview of concolic testing tools and outline their main features. We describe the method of PathCrawler, a concolic testing tool for C programs. Some implementation issues, that are usually omitted, will be described here in detail in order to illustrate the particular problems that this technique raises for an implementation in the cloud. Next, we describe the problems we faced, and provide the solutions we found during the implementation of PathCrawler-online, a prototype of testing service in the cloud whose limited evaluation version is available at (Kosmatov, 2010b). This web service implements all basic features of an online test generation service: uploading a C program to be tested, customizing test parameters and an oracle, generating and executing test cases on the program under test, and providing to the user the generated test cases and test coverage statistics. Next, we present recent results on verification of operating systems and cloud hypervisors. We underline several successful applications of concolic testing illustrating how this testing technique can improve the reliability of cloud environments. We finish by pointing out future work directions and a conclusion.

Testing in the Cloud

Cloud computing is an emerging paradigm (Zhang, Cheng, & Boutaba, 2010). The term *cloud* became popular after Google's CEO Eric Schmidt used the word to describe the business model of providing services across the Internet in 2006. In mid-2009, the U.S. National Institute of Standards and Technology (NIST) gave the following definition of the concept (Mell & Grance, 2011): Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. The essential characteristics of a cloud include on-demand self-service, broad network access, resource pooling, rapid elasticity and measured service.

The key idea behind the term *cloud computing* is the following: computing services are delivered on demand from a remote location rather than residing on one's own desktop, laptop, mobile device, or even on an organization's server. Computing becomes location- and device-independent, in the sense that it does not matter where information is stored nor where computation/processing is taking place.

Three service models of cloud computing were defined (Mell & Grance, 2011):

- In the *Software as a Service (SaaS)* model, the client uses the provider's applications running on a cloud infrastructure.
- *Platform as a Service (PaaS)* offers to the client the capability to create and deploy onto the cloud infrastructure applications created using programming languages, services and tools supported by the provider.
- Finally, in the *Infrastructure as a Service (IaaS)* model, the clients can provision processing, storage, networks, and other fundamental computing resources where they can deploy and run arbitrary software, which can include operating systems and applications.

The client does not manage or control the underlying cloud infrastructure, except limited user-specific application configuration settings and, for IaaS, limited control of some networking components (e.g., host firewalls). The NIST definition also describes four deployment models: private, community, public or hybrid cloud.

Cloud computing started to be used for testing around 2002. The research has focused on techniques for online testing, ranking, automated test case generation, monitoring, simulation, and policy data provenance (Yu et al., 2010). In (Aalst, 2010), *Software Testing as a Service* is defined as a model of software testing used to test an application as a service provided to customers across the Internet. It enables daily operation, maintenance and testing support through web-based browsers, testing frameworks and servers.

The major benefits of cloud-based testing include the following (see e.g. (Parveen & Tilley, 2010; Gao, Bai, & Tsai, 2011)):

- provide an efficient way to obtain a virtual and scalable test environment over a cloud infrastructure,
- share and leverage computing resources using the cloud,
- support testing operations and obtain required computing resources at anytime,
- share and reuse software testing tools,
- use a utility model *(pay-as-you-go)* as a way to charge provided testing services,
- enable large-scale test data and traffic simulation for system testing.

(Candea, Bucur, & Zamfir, 2010) argue that the *Testing as a Service* (TaaS) paradigm will have a significant impact on software engineering. It can be compared to the introduction of high level programming languages and compilers in the 1950s that eliminated most direct use of assembly language and improved the productivity-to-bugs ratio. Another important turning point in software development was the creation of faster hardware and compilers, providing quick feedback on syntax errors and low-level programming errors during the build process. These two events transformed the way programmers' write code: less concern for the minor details and more time devoted to the higher level thought process. Like a modern compiler almost immediately reports *syntax* errors, TaaS can provide quick feedback on *semantic* correctness, that will lead to higher software reliability in general. The authors also describe three forms of TaaS: TaaS_D for developers to more thoroughly test their code, TaaS_H for end users to check the software they install, and TaaS_C certification services that enable consumers to choose among software products based on the products' measured reliability.

However, migrating to the cloud can be costly, and it is not always the best solution to all testing problems. (Parveen & Tilley, 2010) discuss when and how to migrate software testing to the cloud, and in particular, what kinds of programs and what kinds of testing are suitable for testing in the cloud. The characteristics of a program that make it feasible for its testing process to be migrated to the cloud include the following:

1. test cases are independent from one another (or their dependencies are easily identifiable),
2. a self-contained and easily identifiable operational environment, and
3. programmatically accessible interface that is suitable for automated testing.

Among the kinds of testing that could benefit from migrating to the cloud, the authors indicate unit testing, high-volume automated testing and performance testing.

(Riungu, Taipale, & Smolander, 2010b, 2010a) present a recent qualitative study on software testing as an online service from the practitioners' point of view. Based on several interviews with software testing providers and customers, the authors show that the demand for TaaS is on the rise. The underlying research question was: "What conditions influence

software testing as an online service?" Based on the received responses, they discuss the requirements, benefits, challenges, and some research issues from the perspectives of online business vendors and practitioners.

One recent example of testing in the cloud is D-Cloud (Banzai et al., 2010; Hanawa et al., 2010), a large-scale software testing environment for dependable distributed systems. It uses computing resource provided by the cloud to execute several test cases simultaneously, and thus to accelerate software testing process. D-Cloud takes advantage of virtual machine technology to provide a fault injection facility that allows hardware faults to be emulated according to the user's request. It also offers an advanced configuration utility that facilitates the system setup and the testing process.

The York Extensible Testing Infrastructure (YETI) is another example of testing in the cloud (Oriol & Ullah, 2010). YETI is an automated random testing tool for Java with the ability to test programs written in different programming languages. The stand-alone implementation of the tool suffered from two issues, low performances and security problems related to the execution of user-provided code. A new cloud implementation of YETI significantly improves the performances (by testing in parallel) and solves potential security issues (by executing Java classes on clean virtual machines).

(Ciortea, Zamfir, Bucur, Chipounov, & Candea, 2009) introduces Cloud9, a cloud-based testing service that promises to make high-quality testing fast, cheap, and practical. Based on the KLEE testing tool (Cadar, Dunbar, & Engler, 2008), Cloud9 enables parallel symbolic execution for computer clusters operating on public cloud infrastructures such as Amazon EC2 and clusters running cloud software like Eucalyptus. The authors show how to dynamically partition the complete testing task (i.e. the whole program execution tree) into smaller tasks (described as execution sub-trees) run in parallel. The authors currently prepare a public release of Cloud9, that is for the moment available only for identified users. Another facet of testing in the cloud, for testing of the cloud itself, is addressed in (T. M. King & Ganti, 2010).

Reliability and security of the cloud are widely recognized by practitioners and researchers as a vital requirement, both for testing in the cloud (Riungu et al., 2010b; Parveen & Tilley, 2010) and for cloud-based services in general (Zhang et al., 2010). This issue is the main concern of this chapter, that we will address from two points of view, for migrating software testing to the cloud and for testing of the cloud. We will focus on the promising testing techniques presented in the next section.

Test Generation Techniques Combining Concrete and Symbolic Execution

Among other novel testing techniques, various combinations of concrete and constraint-based symbolic execution (see e.g. (Kosmatov, 2010a)) were developed during the last decade. By *concrete execution* we mean the usual execution of a program, for instance, of the compiled binary executable of a C program, where the program is run with a (concrete) input data.

Symbolic execution was introduced in software testing in 1976 by L. A. Clarke (Clarke, 1976) and J. C. King (J. C. King, 1976) in order to reason about executions of a program without running it. The key idea behind it is to use symbolic values, instead of concrete ones, as input values, and to represent the values of program variables as symbolic expressions. As a result, the outputs computed by a program are expressed in terms of the symbolic inputs. Symbolic execution is often performed with help of constraints that represent particular conditions on symbolic input values under which program execution follows a particular path or, more generally, verifies some property at some program point. Test generation is performed by solving the corresponding constraints, using a decision procedure or constraint solver, and produces a set of test cases. A *test case* contains an input data for the program under test, and may contain the expected result of its execution.

For example, in the following C program

```
1   //returns minimum of X, Y
2   int min(int X, int Y){
3     if( X > Y )
4       return Y;
5     else
6       return X;
7   }
```

line 4 is executed if and only if the inputs X and Y satisfy the constraint $X > Y$, and in this case the program returns Y. This is an example of symbolic reasoning. For the constraint $X > Y$, a constraint solver finds a solution that represents the input data of a test case, say, $X = 15$, $Y = 7$. The program can be (concretely) executed by this test case and will return 7, a concrete value.

Symbolic execution can also be used for program debugging, where it checks for runtime errors or assertion violations and generates test inputs that trigger those errors. We present constraint-based symbolic execution in detail in the next section.

Combined concrete/symbolic techniques, with various modifications, were successfully applied for implementation of several testing tools for *white-box testing*, that is, for the testing approach in which the implementation code is examined for designing tests. These techniques are often called *dynamic symbolic execution*, or *concolic* (CONCrete/symbOLIC) testing. The first concolic tools for C programs, PathCrawler (Williams, Marre, Mouy, & Roger, 2005) and DART (Godefroid, Klarlund, & Sen, 2005), perform symbolic execution while the program is executed on some concrete input values. DART uses concrete values and randomization to simplify the constraints when it cannot reason precisely. UIUC's CUTE (Sen, Marinov, & Agha, 2005) (for C) and jCUTE (for Java) extend DART to handle multi-threaded programs that manipulate dynamic data structures using pointer operations. CUTE represents and solves pointer constraints approximately. In multi-threaded programs, CUTE systematically generates both test inputs and thread schedules by combining concolic execution with dynamic partial order reduction.

Stanford's EXE (Cadar, Ganesh, Pawlowski, Dill, & Engler, 2006) is designed for testing complex software, including systems code, so it accurately builds bit-level constraints for C

expressions, including those involving pointers, casting, unions, and bit-fields. It performs mixed concrete/symbolic execution where the concrete state is maintained as part of the normal execution state of the program. A new version of the EXE tool, KLEE (Cadar et al., 2008) stores a much larger number of concurrent states, employs a variety of constraint solving optimizations, and uses search heuristics to get high code coverage.

NASA's Symbolic (Java) PathFinder (Anand, Pasareanu, & Visser, 2007) analyzes both Java bytecode and statechart models, e.g., Simulink/Stateflow, Standard UML, Rhapsody UML, etc., via automatic translation into bytecode. It handles mixed integer and real constraints, as well as complex mathematical constraints by means of solving heuristics.

Microsoft's PEX (Tillmann & Halleux, 2008) implements dynamic symbolic execution to generate test inputs for .NET code, supporting languages such as C#, VisualBasic, and F#. PEX uses concrete values to simplify constraints and treats almost all .NET instructions symbolically, including safe and unsafe code, as well as instructions that refer to the object oriented .NET type system, such as type tests and virtual method invocations. PEX supports primitive and (recursive) complex data types, for which it automatically computes a factory method that creates an instance of a complex data.

Another tool from Microsoft, SAGE (Godefroid, Levin, & Molnar, 2012) implements *automated whitebox fuzzing*, a modified concolic technique for symbolically executing very long execution traces with billions of instructions, for symbolic execution at the x86 assembly level and for compact representation of path constraints.

UC Berkeley's CREST (Burnim & Sen, 2008) is an open-source extensible platform for building and experimenting with concolic search heuristics for selecting which paths to test for programs with far too many execution paths to exhaustively explore.

Unlike other concolic tools, PathCrawler aims for complete coverage of a certain class of programs rather than for incomplete coverage of any program. It runs the program under test on each test case in order to recover a trace of the execution path. However, in PathCrawler's case concrete execution is used merely for reasons of efficiency and to demonstrate that the test does indeed activate the intended execution path. PathCrawler does not approximate constraints for program instructions that it cannot treat since these results can only provide an incomplete model of the program's semantics.

In the next section, we present an advanced unit testing technique combining constraint-based symbolic execution and concrete execution of the program under test. This is essentially the technique implemented by the PathCrawler tool. We consider the *all-path test coverage criterion* which requires to generate a set of test cases such that every possible execution path of the program under test is executed by at least one test case. Since this criterion is very strong, weaker path-oriented criteria were proposed, requiring to cover only paths of limited length, or with limited number of loop iterations, etc. Test generation methods for these weaker criteria are obtained from the all-path test generation method by slight modifications that do not impact the main idea of the method and its deployment online.

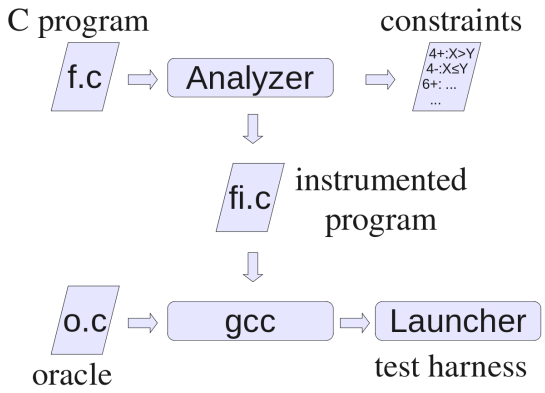


Figure 1: PathCrawler Analyzer

PathCrawler Methodology

Description of the PathCrawler method

The PathCrawler tool is basically composed of three main modules that we call Analyzer, Path explorer and Solver. PathCrawler uses COLIBRI, an efficient constraint solver developed at CEA LIST and shared with two other testing tools: GATeL (Marre & Arnould, 2000) and OSMOSE (Bardin & Herrmann, 2008). COLIBRI provides a variety of types and constraints (including non-linear constraints), primitives for labelling procedures, support for floating point numbers and efficient constraint resolution (Leconte & Berstel, 2006; Gotlieb, Leconte, & Marre, 2010; Michel, 2002; Marre & Michel, 2010). Experiments in (Bardin, Herrmann, & Perroud, 2010) using SMT-LIB benchmarks show that COLIBRI can be competitive with powerful SMT solvers.

The PathCrawler method contains two main stages. The first stage is illustrated by Figure 1. The user provides the source code of the C program under test, in one or several files, denoted here for simplicity by `f.c`, and indicates the function to be tested.

Analyzer parses the program `f.c`, generates its instrumented version and translates its statements into an intermediate representation in constraints (see Figure 1). These constraints encode all the information on the control and data flow of the program (assignments, branches, loops, function calls, etc.) necessary for symbolic execution at the second stage. The instrumented version, denoted here by `fi.c`, contains the original source code of `f.c` enriched with path tracing instructions. The compilation of this instrumented version, performed here with `gcc`, provides an executable test harness that we call Launcher. Launcher runs the program under test with a given input and records the program path that was activated.

The user may also provide C code with an optional oracle, denoted here by `o.c`, that checks if the result of the program executed by a given test case is correct or not. When provided, the oracle is loaded into Launcher, and called after the execution of a test case to produce a verdict on the results.

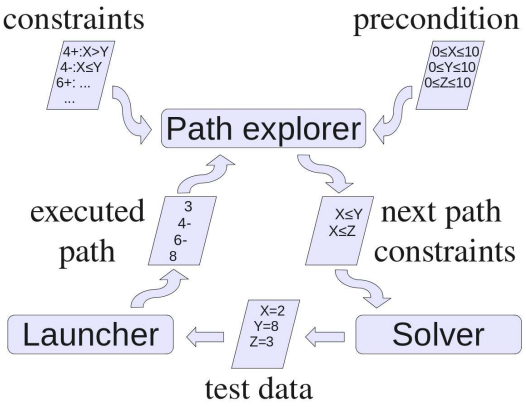


Figure 2: PathCrawler Generator

Figure 2 illustrates the second stage where a depth-first exploration algorithm with constraint-based symbolic execution (in Path explorer) and a constraint solver are used to generate test cases. This stage uses the module Launcher and the intermediate program representation in constraints obtained at the first stage. In addition, the user defines a *precondition*, i.e. the conditions on the program's input for which the behavior is defined. Any generated test case must satisfy the precondition. For example, a typical precondition for a function computing a square root of x will be $x \geq 0$ (unless we intentionally want to check the behavior for negative numbers). A precondition for dichotomic search of a given element in a given sorted array should specify that the input array is sorted and may restrict the intervals of values for the size and the elements of the array.

At this stage, three modules are used one after another in a loop: Path explorer, Solver and Launcher. In the beginning of each iteration, Path explorer determines the next (partial) program path to be covered, and sends the corresponding path constraints to Solver. Solver generates test data satisfying the constraints and sends it to Launcher. Launcher executes the program under test on the test data and sends the executed program path to Path explorer, which starts the next iteration.

When Solver fails to generate a test case (e.g. for an infeasible program path), the method skips the concrete execution step with Launcher and goes directly to Path explorer for a next path choice. When Path explorer has no more paths to cover, all paths are covered and test generation stops. In the first iteration, Path explorer starts with the empty partial path. In the following iterations, Path explorer explores remaining program paths in a depth-first search.

A running example

Let us illustrate the method on a running example. Consider the program `min3` and its control flow graph (CFG) shown in Figure 3. The function `min3` takes three integer parameters `X`, `Y`, `Z`, and returns the minimum among them. The logical variables X, Y, Z will denote the values of the input parameters. For simplicity, we restrict the domains of the parameters

```
1   //returns minimum of X, Y, Z
2   int min3(int X, int Y, int Z){
3     int min = X;
4     if( min > Y )
5       min = Y;
6     if( min > Z )
7       min = Z;
8     return min;
9   }
```

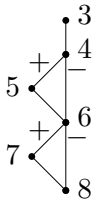


Figure 3: Function min3 returning the minimum of its three arguments, and its CFG

to $[0, 10]$, so the precondition is defined by:

$$0 \leq X \leq 10,\ 0 \leq Y \leq 10,\ 0 \leq Z \leq 10.$$

The CFG illustrates possible execution paths of the program. We denote an execution path by a sequence of line numbers of its statements, e.g. $3, 4^-, 6^+, 7, 8$. A decision is denoted by the line number of the condition followed by a "+" if the condition is satisfied, and by a "−" otherwise.

The first stage of the method creates a constraint representation of the program and the module Launcher (cf Figure 1). Let us describe in more detail the test generation loop of the second stage illustrated by Figure 4. The first iteration starts with Path explorer that sends to Solver the path constraints for the empty path ϵ, i.e. just the precondition constraints, denoted by $\langle \text{precond} \rangle$. In other words, it asks for any test data satisfying the precondition. Solver generates Test 1, say, $X = 3,\ Y = 7,\ Z = 2$. Next, Test 1 is executed by Launcher and the executed path is sent to Path Explorer (see the right column boxes of Figure 4).

Path explorer finds, in a depth-first search, the next partial program path to be covered and symbolically executes it. Technically, it takes the previously explored path, negates the last not-yet-negated branch on this path and drops the statements that follow it.

In the second iteration, the last not-yet-negated decision in the path $3, 4^-, 6^+, 7, 8$ is 6^+, so the next partial path to be covered is $3, 4^-, 6^-$. Path explorer maintains a representation of the program memory state at every moment of symbolic execution and uses it to produce a constraint solving problem (see the left column boxes of Figure 4) corresponding to the selected program path, so that any solution of this problem gives a test case exercising the desired path. The constraints are expressed in terms of program inputs. Then Solver generates Test 2, and Launcher executes it on the instrumented version of the program under test to obtain the exercised path $3, 4^-, 6^-, 8$.

Since the last not-yet-negated condition of this path is now 4^-, in the third iteration Path explorer defines the next partial path to be covered as $3, 4^+$ and constructs the corresponding constraints. Next, Solver generates Test 3, which exercises the path $3, 4^+, 5, 6^-, 8$.

The last not-yet-negated condition of this path being 6^-, in the fourth iteration Path explorer asks to cover the partial path $3, 4^+, 5, 6^+$. Notice that the constraint $Y > Z$ takes into

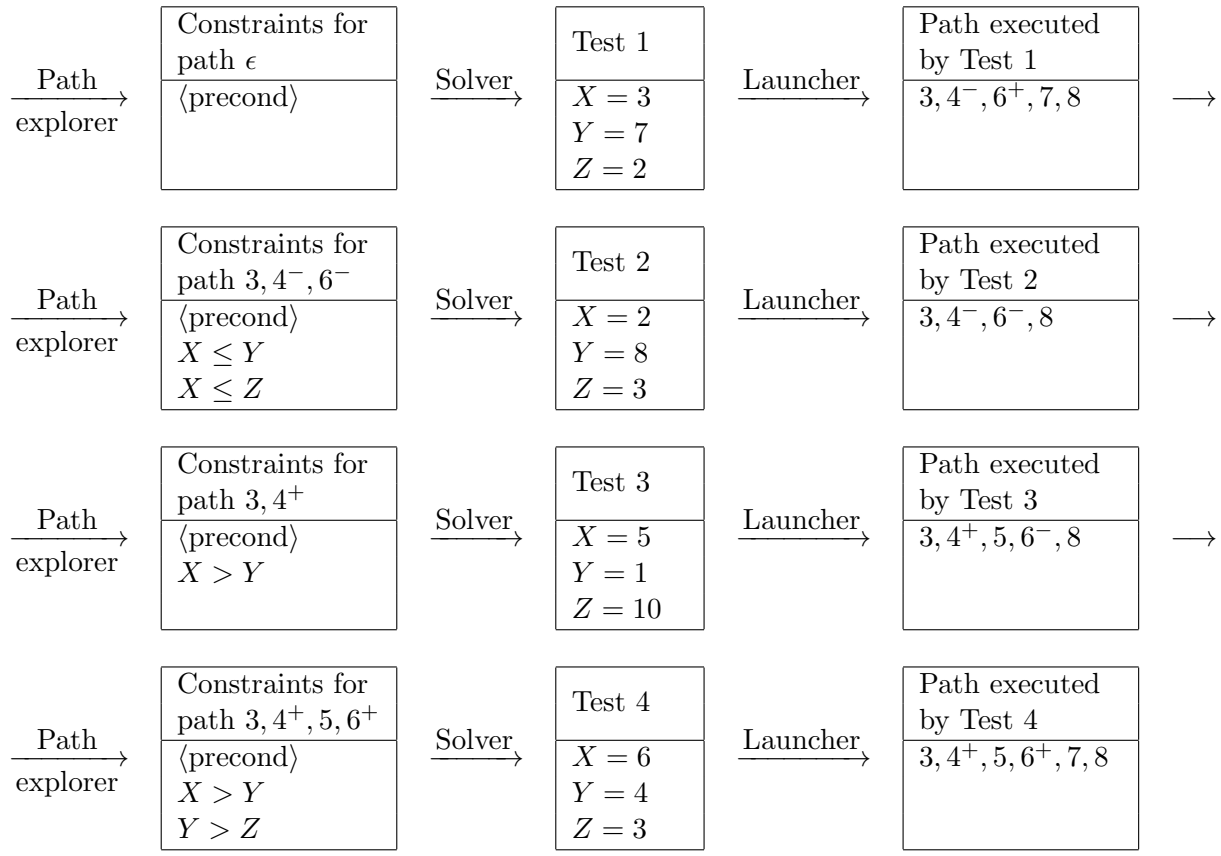


Figure 4: Test generation for the function min3 of Figure 3

account the current value of the variable `min` at the decision 6^+ modified by the assignment line 5. Next, Test 4 is generated and executed.

Since all decisions of the executed path $3, 4^+, 5, 6^+, 7, 8$ have been already negated, all possible paths have been explored and the test generation process stops. The four generated tests cover the four feasible program paths of the program in Figure 3.

Deployment in the Cloud: Problems and Solutions

This section describes the problems we encountered during the online deployment of the PathCrawler test generation tool and the solutions we found and implemented. We discuss several important risks that should be taken into account while creating the architecture of a similar testing service in the cloud, or more generally, in PaaS clouds like Google App Engine (Google, 2012) where the code submitted by the user runs in the cloud. To some extent, these risks affect any deployment of concolic testing, from a local one to that in the cloud. However, the risks are all the greater for a publicly available web service given its larger accessibility for users all over the world. The solutions we propose preserve the efficiency of the method, that is another important requirement for testing in the cloud. We also present

the current state, interface and restrictions of our implementation of PathCrawler-online.

Malicious code and concrete execution

The main problem is related to the concrete execution of the program under test. The submitted C source code to be tested can do almost anything, and it seems extremely difficult to correctly distinguish a malicious behavior from an intended regular use of the same features. Therefore, executing in a shared environment in the cloud a potentially malicious program raises important security problems. This program may be a malware or a spyware intentionally submitted to harm or to discover the server or network configuration or data. It may also contain an unintentional erroneous behavior that can overload available resources or destabilize the server, executed programs or other clients' sessions.

By definition of software testing, any testing approach requires execution of the program under test on the selected test data. However, test generation and test execution can often be separated and need not to be run in the same environment. For example, in model-based testing, test cases are derived from the model without any knowledge of the implementation's source code (see e.g. (Kosmatov, 2010a)). Therefore, the test generation step and the execution of the program on the generated tests can easily be separated: the test generation utility can be deployed in the cloud, while the test execution remains on the client's side or runs in a dedicated, highly secured environment. Similarly, in white-box testing approaches using symbolic execution only, concrete execution of the client's source code is not necessary at the test generation step.

The specificity of PathCrawler-like methods is that concrete execution of the client's code is closely integrated into the second stage of the method. As we saw in Figure 2, the program under test is executed by Launcher for each test case found during the test generation process.

One possible way to avoid concrete execution of potentially dangerous code in the cloud consists of sending Launcher to the client, running it on the client's network and keeping Analyzer, Path Explorer and Solver in the cloud. However, this solution needs permanent information exchange between the test generator (with Path Explorer and Solver) and the distant execution of Launcher on the client's side. Hence, this solution results in considerable efficiency loss and is not acceptable. Notice that the efficiency of combined PathCrawler-like methods is actually due to the speed of concrete execution, replying almost immediately when running on the same server without any communication slow-down.

Another possible solution consists in executing all three generator modules (Path Explorer, Solver and Launcher) in the client's network, but it essentially requires to install the tool on the client's side, that obviously does not correspond to the objectives of a deployment in the cloud.

How to ensure secure execution of the client's code in the cloud without diminishing the performances of the method? The solution we used in our implementation proposes to split the complete method into two parts, secure and insecure ones, and to run the potentially insecure steps (including concrete execution of the client's code) in a virtual machine. In our presentation in the previous section we split the method into two stages that correspond

exactly to this requirement. The first stage shown in Figure 1 analyzes and transforms the source code under test without executing it, so it can be considered secure. It is therefore more efficient to keep this part outside the virtual machine that isolates concrete execution. The second stage illustrated in Figure 2 executes the compiled instrumented version of the submitted program, so it runs the risk of malicious behavior. We isolate this stage in a separate virtual machine.

Particular reliability and security risks

Isolating a potentially dangerous process in a virtual machine is nowadays a well-known paradigm that has been implemented in various trustworthy technical solutions. Many of them have reliable support and quite satisfactory performances. For security reasons we do not communicate on the particular virtualization mechanism we used in PathCrawler-online.

Let us present several particular risks that we studied to secure our implementation, and that must be kept in mind during any implementation of a similar testing service.

The submitted program may read, modify or destroy files on the system. The virtual machine will protect from unauthorized access to the files outside, but a limited information exchange with outside files has to be allowed in some way in order to transfer into the virtual machine the files related to the desired test generation task and to extract the results. A typical error would be to allow broad access to files of the underlying machine such as to a disk containing other sensitive data or used by other processes. Indeed, the program under test can try to modify the accessible files, or read and encode the discovered data in test generation results. The privileges of the users executing the virtual machine and the program under test inside it must be carefully checked and restricted.

The submitted program may saturate disk space on the system. The program under test can fill up the disk producing a great amount of information, or provoke generation of huge test data files, that can result in lack-of-space problems in the system. Therefore, it will be dangerous to allow access even to a subdirectory of a disk containing sensitive data or used by other processes, since saturating this disk can lead to negative consequences. We think the best way to set up information exchange with the virtual machine is a separate, dedicated partition. At the same time, we periodically check the size of written files, including those with test generation results.

The submitted program may contain a fork bomb. This is a well-known denial-of-service attack that should be explicitly checked and excluded. A *fork bomb* is a program creating a large number of processes very quickly in order to saturate available system resources, for instance, like in

```
while( 1 ) {  fork(); }
```

that infinitely tries to copy itself into a new process. This risk can be easily prevented by setting a limit on the number of processes that one user may own, without completely forbidding the `fork` operation.

The submitted program may try to overload other available resources (processor, RAM, etc.). This risk is normally prevented by using the virtual machine, which should be allowed to use only limited resources. However, it is still worth testing.

The submitted program may maliciously send or receive data using the Internet. This is definitely the risk the most difficult to prevent, since we can neither easily detect a malicious behavior on the Internet nor protect distant machines from such a behavior. In our implementation, we deactivate network access in the virtual machine and do not treat programs under test using Internet. In our opinion, the number and variety of various attacks using Internet today makes it improbable to provide a reliable public testing service in the cloud for software using Internet.

Security of testing in the cloud was also addressed in YETI (Oriol & Ullah, 2010), in which virtual machines are used to secure test execution. However, the problem of splitting the method into secure and insecure parts does not occur in YETI since it implements random testing, in which test generation and test execution steps are not so closely tied as in PathCrawler.

The issues described in this section are related to those encountered in commercial PaaS clouds where the user code is executed in the cloud. For example, Google App Engine (Google, 2012) also uses virtualization to secure execution, but its limitations are different from PathCrawler-online. C code is not allowed to be executed in Google App Engine, while PathCrawler is designed for testing C code. In Google App Engine, applications cannot write to the file system in any of the runtime environments, while PathCrawler-online allows read-write access to files inside the virtual machine, that is necessary e.g. to write test generation results. Google App Engine forbids creating sub-processes while PathCrawler-online allows a limited number of sub-processes to be opened by the program under test. Google App Engine also restricts access to and from the Internet, that is not allowed in PathCrawler-online.

Online test generation with PathCrawler

PathCrawler-online (Kosmatov, 2010b) provides a web service with all basic features of an online testing tool:

- uploading a complete multi-file C project and an oracle,
- generating default test parameters,
- customizing test parameters, such as input variables, precondition, test generation strategy,
- generating test cases in XML format,
- executing them on the program under test in a secure environment,
- generating C source code of ready-to-use test drivers that allow to execute the program under test on each test case on the client's testing environment,

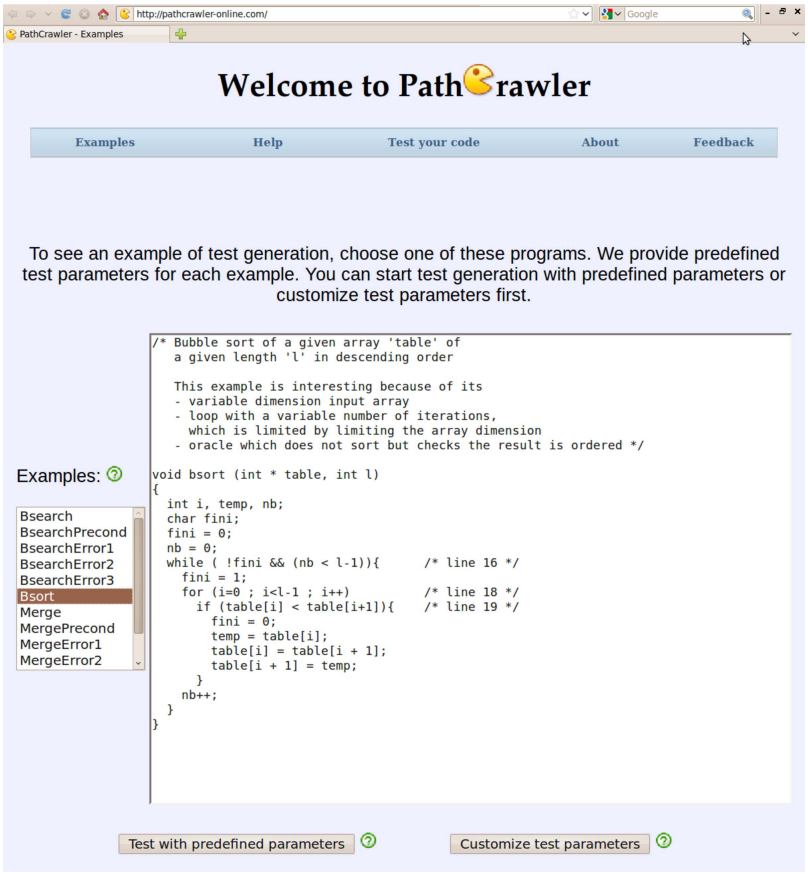
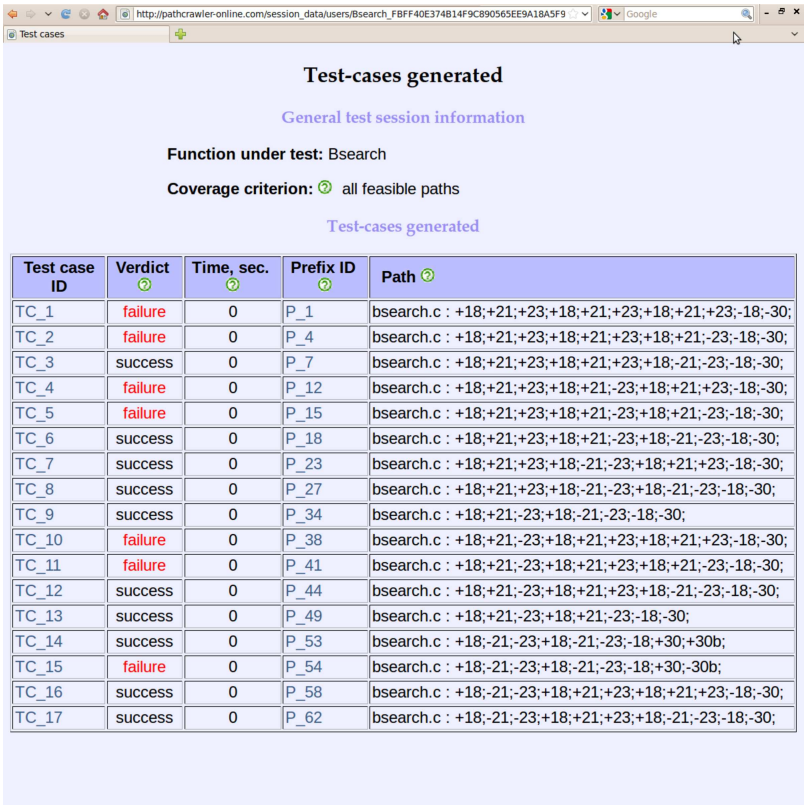


Figure 5: PathCrawler Online home page

- providing to the user the test cases, test execution and coverage statistics on-the-fly in a web browser.

Figure 5 shows the initial page of the service where the user can select one of the available examples and start test generation. Another page allows to upload an archive with the user's C project. It is possible to use default precondition, testing strategy and oracle, or to customize them first. After the test generation, the user can see the summary of the test session, the list of all explored paths and the list of test cases. Figure 6 shows an example of generated test cases, some of which fail (the failure verdict appears in red). For each test case, a separate page shows the generated test data, the executed path, its predicate (also called *path condition*), concrete values of the output variables, their symbolic expression in terms of inputs, the oracle's verdict and the generated test driver.

This free evaluation version does not allow the user to identify himself, to save a test generation session, to continue it later, to use a makefile or customized compilation options and to download the generated test cases and statistics in XML format. Besides, the current version sets restrictive bounds for the number of explored paths, the number of generated test cases, test generation time and disk space for generated files that cannot be overpassed.



Test-cases generated

General test session information

Function under test: Bsearch

Coverage criterion: all feasible paths

Test-cases generated

Test case ID	Verdict	Time, sec.	Prefix ID	Path
TC_1	failure	0	P_1	bsearch.c : +18;+21;+23;+18;+21;+23;+18;+21;+23;-18;-30;
TC_2	failure	0	P_4	bsearch.c : +18;+21;+23;+18;+21;+23;+18;+21;-23;-18;-30;
TC_3	success	0	P_7	bsearch.c : +18;+21;+23;+18;+21;+23;+18;-21;-23;-18;-30;
TC_4	failure	0	P_12	bsearch.c : +18;+21;+23;+18;+21;-23;+18;+21;+23;-18;-30;
TC_5	failure	0	P_15	bsearch.c : +18;+21;+23;+18;+21;-23;+18;+21;-23;-18;-30;
TC_6	success	0	P_18	bsearch.c : +18;+21;+23;+18;+21;-23;+18;-21;-23;-18;-30;
TC_7	success	0	P_23	bsearch.c : +18;+21;+23;+18;-21;-23;+18;+21;+23;-18;-30;
TC_8	success	0	P_27	bsearch.c : +18;+21;+23;+18;-21;-23;+18;-21;-23;-18;-30;
TC_9	success	0	P_34	bsearch.c : +18;+21;-23;+18;-21;-23;-18;-30;
TC_10	failure	0	P_38	bsearch.c : +18;+21;-23;+18;+21;+23;+18;+21;+23;-18;-30;
TC_11	failure	0	P_41	bsearch.c : +18;+21;-23;+18;+21;+23;+18;+21;-23;-18;-30;
TC_12	success	0	P_44	bsearch.c : +18;+21;-23;+18;+21;+23;+18;-21;-23;-18;-30;
TC_13	success	0	P_49	bsearch.c : +18;+21;-23;+18;+21;-23;-18;-30;
TC_14	success	0	P_53	bsearch.c : +18;-21;-23;+18;-21;-23;-18;+30;+30b;
TC_15	failure	0	P_54	bsearch.c : +18;-21;-23;+18;-21;-23;-18;+30;-30b;
TC_16	success	0	P_58	bsearch.c : +18;-21;-23;+18;+21;+23;+18;+21;+23;-18;-30;
TC_17	success	0	P_62	bsearch.c : +18;-21;-23;+18;+21;+23;+18;-21;-23;-18;-30;

Figure 6: PathCrawler Online test generation results: test cases

Towards Reliable Cloud Environments

Cloud computing nowadays becomes pervasive in many domains. A substantial expansion of mobile devices will make it all the more popular in the future. As we mentioned before, reliability and security of cloud environments are currently major challenges for research and industry (Zhang et al., 2010).

In this section, we describe some of the most representative results of the past five years in this domain, and show the promising role that concolic testing based techniques can play for this goal. We focus on cloud hypervisors, that virtualise the underlying architecture, allowing a number of guest machines to be run on a single physical host. Hypervisors represent an interesting and challenging target for software verification because of their critical and complex functionalities. They can be *hosted* by the operating system of the physical host, or *native,* that is, running directly on the physical host to control the hardware and to manage guest operating systems. Therefore recent contributions to verification of operating systems in general are also relevant.

A recent trend in verification of operating systems and cloud hypervisors is formal verification (Loulergue, Gava, Kosmatov, & Lemerre, 2012), that includes a formal machine-checked proof that the program satisfies its specification. The *specification* (or *contract*) describes the expected behavior of each function and is often expressed by logical formulae associ-

ated to the source code. The machine-checked mathematical proof guarantees that the code respects the contract.

A recent work (Klein et al., 2009) presented rigorous, formal verification for the OS microkernel seL4. The proof being complete, this work allows devices running seL4 to achieve the highest assurance level, such as the EAL7 evaluation level "formally verified design and tested" of the Common Criteria for Information Technology Security Evaluation (CCRA, 2012), a modern standard for software security evaluation. Another formal verification of a microkernel was described in (Alkassar, Paul, Starostin, & Tsyban, 2010). In both cases, the verification used interactive machine-checked proof with the theorem prover Isabelle/HOL (Nipkow, Paulson, & Wenzel, 2002). Although interactive theorem proving requires human intervention to construct and guide the proof, it has the benefit to serve a general range of properties and is not limited to specific properties treatable by more automated methods of verification such as static analysis or model checking.

The formal verification of a simple hypervisor in (Alkassar, Hillebrand, Paul, & Petrova, 2010) uses VCC (Cohen et al., 2009), an automatic first-order logic based verifier for C. The underlying system architecture is precisely modeled in VCC, and the system software is then proved correct. Unlike (Klein et al., 2009) and (Alkassar, Paul, et al., 2010), this technique uses automated theorem proving methods, but writing the specification remains manual.

(Alkassar, Cohen, Kovalev, & Paul, 2012) reports on verification of TLB (translation lookaside buffer) virtualization, a core component of modern hypervisors. Because devices run in parallel with software, they typically necessitate concurrent program reasoning even for single-threaded software. The authors give a general methodology for verifying virtual device implementations, and demonstrate the verification of TLB virtualization code in VCC.

Does it mean that reliability of cloud environments will be formally verified in the very near future, excluding any risk of error or attack? Despite the spectacular progress in application of formal verification methods to the cloud, fully, formally verified cloud environments are unlikely to become the reality of cloud computing over the next years.

Indeed, formal verification still has important limitations. First of all, it is very costly. According to (Klein, 2010), the cost of the verification of the seL4 microkernel was around 25 person-years, and required highly qualified experts. seL4 contains only about 10,000 lines of C code, and verification cost is about $700 per line of code. Even if these techniques may optimistically scale up to 100,000s lines of code for appropriate code bases, realistic systems with millions of lines of code still seem out of reach for the near future.

Second, formal verification of a microkernel or a hypervisor remains valid only for a particular version being verified. Therefore, any evolution of the software can potentially alter the verified properties, and requires new verification.

Third, the verification is currently performed for the source code with several assumptions (Klein, 2010). It assumes that the C compiler and linker are correct. Although first *certified compilers,* whose correctness for some supported processors is formally established, have been developed (Leroy, 2009), they are not yet commonly used in industry. The verification in (Klein et al., 2009; Alkassar, Hillebrand, et al., 2010) assumes sequential execution, and its extension to a concurrent hypervisor seems very challenging. The verification also

assumes correctness of hardware, cache-flushing instructions, and boot code. Some specific hardware failures, for instance provoked by overheating, or hardware details beneath the lowest verification level can still be attacked. Therefore, "you still need orthogonal methods to ensure that you are building the right system with the right requirements, and not the wrong system correctly" (Heiser, Murray, & Klein, 2012).

Among such orthogonal methods, structural software testing techniques based on concrete/symbolic execution are of great interest for testing of the cloud. They do not have the three aforementioned limitations. First, they are highly automated, so their cost is almost negligible compared to formal verification. Second, an updated version of software can be easily tested again without human intervention. Finally, testing in the realistic cloud environment takes into account all real-life environment details related to hardware, boot code, compiled code, etc. Moreover, concolic testing can efficiently benefit from a cloud infrastructure since it can be easily parallelized (Ciortea et al., 2009). Another strength of concolic testing is its capacity to achieve a high level of structural test coverage, that aims at activating by some tests all program statements, or all program branches, or all program paths (of limited length), etc. Structural coverage is required by many software evaluation standards, for example, the Common Criteria (CCRA, 2012), DO-178B (in the avionics domain), ECCS-E-ST-40C (space), ISO 26262 (automotive), EC/EN 61513 (nuclear), etc.

One of the most encouraging examples of concolic testing is reported in (Godefroid et al., 2012). Based on modified concolic testing, the SAGE tool has discovered many security-related bugs in many large Microsoft applications, notably during the development of Windows 7. Finding these bugs has saved Microsoft millions of dollars by avoiding security patches to more than one billion PCs. Since 2008, SAGE has been running 24/7 on approximately 100 machines automatically testing hundreds of applications in Microsoft security testing labs. SAGE is also able to test large applications, where it can find bugs resulting from problems across multiple components. (Godefroid & Molnar, 2010) argue that fuzz testing in the cloud will revolutionize security testing.

Another concolic tool, EXE (Cadar et al., 2006) discovered deep bugs and security vulnerabilities in a variety of complex code, ranging from library code to UNIX utilities, file systems, packet filters, and network tools. Its successor KLEE (Cadar et al., 2008) was applied to 452 applications (over 430K total lines of code) including GNU COREUTILS, containing roughly 80,000 lines of library code and 61,000 lines in the actual utilities, and the HiStar OS kernel. KLEE found 56 serious bugs, some of which had been missed for over 15 years.

As long as complete formal verification of realistic cloud environments remains impossible, software testing keeps its place for assuring their reliability and security. (Klein, 2010) proposes to design the whole system in such a way that its critical components are small and amenable to formal verification. Non critical components will be assumed to do anything they are allowed to subvert the system and will not be formally verified. The security of the system is then reduced to the security mechanisms of the kernel and the behavior of the critical components only. Concolic testing remains a promising technique for testing software components that cannot be formally verified in the real-life environment.

Future Work

Future work includes the development of a complete web service suitable for industrial software testing. We have already implemented the core of such a service in our current prototype PathCrawler-online, but systematic use in industrial software engineering requires additional routines, such as secure data transfer, client identification and account management, test session history, test results download, facilities for integration into the clients' test execution environment, deployment in a reliable cloud infrastructure, etc.

Concolic test generation techniques are also used as part of recent combined methods with other verification techniques, for instance, with partition refinement (Gulavani, Henzinger, Kannan, Nori, & Rajamani, 2006), or with value analysis and program slicing (Chebaro, Kosmatov, Giorgetti, & Julliand, 2012). Future work also includes implementation of a web service for combined verification techniques such as the SANTE method (Chebaro et al., 2012) combining static analysis and PathCrawler-like test generation.

Applying concolic testing in complement to formal verification for verification of the cloud is another promising future work direction that will help to create reliable cloud environments.

Conclusion

Using cloud computing to make software testing services available online and to share them between different companies is an attractive perspective for modern software engineering. In this chapter we addressed security of testing in the cloud for concolic testing, an advanced structural testing technique proposed in the mid 2000s. We reported on PathCrawler-online, a recent implementation of a concolic testing web service. We showed why this kind of testing techniques is particularly challenging for deployment in the cloud. We examined the risks encountered when potentially malicious user code is run in the cloud, and showed how a concolic testing technique can be decomposed into secure and insecure parts and how the insecure part can be isolated using virtualization.

Our experience shows that these technical problems can be successfully resolved. Since its deployment in 2010, PathCrawler-online was used thousands of times. The initial objective of this limited evaluation version has been fully achieved. PathCrawler-online is appreciated by our academic and industrial partners as a convenient tool for discovering concolic test generation with PathCrawler, demonstrating its capabilities and teaching testing in several universities (Kosmatov, Williams, Botella, Roger, & Chebaro, 2012). Initially demonstrated at CSTVA 2011 (Kosmatov, Botella, Roger, & Williams, 2011), it is now frequently used for tutorials on structural unit testing at international events (TAP 2012, TAROT 2012, QSIC 2012, ASE 2012) where several dozens of participants simultaneously run test generation online.

This experience showed an increasing interest to automatic test generation and encouraged us to continue the development of automatic test generation tools and their online versions. Although concolic testing is still a young technique, it has an interesting potential

for software verification, both for testing in the cloud and for testing of the cloud. The advantages of concolic testing include automation, easy replay for a new version of software, parallelization, low cost and the possibility to test a program in its real-life environment. Concolic testing can easily achieve high structural test coverage, that is required by many modern software verification standards. Even if it cannot guarantee the absence of errors, its advantages make of concolic testing a promising complement to formal verification of cloud environments. We believe that concolic testing in the cloud will provide new reliable and cost-efficient solutions that will become the reality of software testing tomorrow.

Acknowledgments

The author would like to thank Bernard Botella, Fabrice Derepas, Matthieu Lemerre, Frédéric Loulergue, Bruno Marre, Benjamin Monate, Muriel Roger and Nicky Williams for their advice and useful discussions, as well as the editors and anonymous referees for lots of valuable remarks and suggestions.

References

Aalst, L. van der. (2010). *Software testing as a service (STaaS)* (Tech. Rep.). Vianen, the Netherlands: Sogeti. (http://www.leovanderaalst.nl/Software Testing as a Service - STaaS.pdf)

Alkassar, E., Cohen, E., Kovalev, M., & Paul, W. J. (2012, January). Verification of TLB virtualization implemented in C. In *the 4th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2012)* (pp. 209–224). Philadelphia, PA, USA: Springer.

Alkassar, E., Hillebrand, M. A., Paul, W. J., & Petrova, E. (2010, August). Automated verification of a small hypervisor. In *the Third International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2010)* (pp. 40–54). Edinburgh, UK: Springer.

Alkassar, E., Paul, W., Starostin, A., & Tsyban, A. (2010, August). Pervasive Verification of an OS Microkernel. In *the Third International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2010)* (pp. 71–85). Edinburgh, UK: Springer.

Anand, S., Pasareanu, C. S., & Visser, W. (2007, March). JPF-SE: A symbolic execution extension to Java PathFinder. In *the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)* (pp. 134–138). Braga, Portugal: Springer.

Banzai, T., Koizumi, H., Kanbayashi, R., Imada, T., Hanawa, T., & Sato, M. (2010, May). D-cloud: Design of a software testing environment for reliable distributed systems using cloud computing technology. In *the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid 2010)* (pp. 631–636). Melbourne, Victoria, Australia: IEEE Computer Society.

Bardin, S., & Herrmann, P. (2008, April). Structural testing of executables. In *the First IEEE International Conference on Software Testing, Verification, and Validation (ICST'08)* (pp. 22–31). Lillehammer, Norway: IEEE Computer Society.

Bardin, S., Herrmann, P., & Perroud, F. (2010, March). An alternative to SAT-based approaches for bit-vectors. In *the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010)* (pp. 84–98). Paphos, Cyprus: Springer.

Burnim, J., & Sen, K. (2008, September). Heuristics for scalable dynamic test generation. In *the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)* (pp. 443–446). L'Aquila, Italy: IEEE.

Cadar, C., Dunbar, D., & Engler, D. R. (2008, December). KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)* (pp. 209–224). San Diego, California, USA: USENIX Association.

Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., & Engler, D. R. (2006, November). EXE: automatically generating inputs of death. In *the 13th ACM Conference on Computer and Communications Security (CCS'06)* (pp. 322–335). Alexandria, Virginia, USA: ACM Press.

Candea, G., Bucur, S., & Zamfir, C. (2010, June). Automated software testing as a service. In *the First ACM Symposium on Cloud Computing (SoCC 2010)* (pp. 155–160). Indianapolis, Indiana, USA: ACM Press.

CCRA. (2012). *The Common Criteria for Information Technology Security Evaluation.* (http://www.commoncriteriaportal.org)

Chebaro, O., Kosmatov, N., Giorgetti, A., & Julliand, J. (2012, March). Program slicing enhances a verification technique combining static and dynamic analysis. In *the 27th Annual ACM Symposium On Applied Computing (SAC 2012)* (pp. 1284–1291). Riva del Garda, Italy: ACM Press.

Ciortea, L., Zamfir, C., Bucur, S., Chipounov, V., & Candea, G. (2009). Cloud9: a software testing service. *Operating Systems Review*, *43*(4), 5–10.

Clarke, L. A. (1976). A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, *2*(3), 215–222.

Cohen, E., Dahlweid, M., Hillebrand, M. A., Leinenbach, D., Moskal, M., Santen, T., et al. (2009, August). VCC: a practical system for verifying concurrent C. In *the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)* (pp. 23–42). Munich, Germany: Springer.

Gao, J., Bai, X., & Tsai, W.-T. (2011). Cloud testing - issues, challenges, needs and practice. *Software Engineering : An International Journal (SEIJ)*, *1*(1), 9–23.

Godefroid, P., Klarlund, N., & Sen, K. (2005, June). DART: Directed automated random testing. In *the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)* (pp. 213–223). Chicago, IL, USA: ACM Press.

Godefroid, P., Levin, M. Y., & Molnar, D. A. (2012). SAGE: whitebox fuzzing for security testing. *Commun. ACM*, *55*(3), 40–44.

Godefroid, P., & Molnar, D. (2010, March). *Fuzzing in The Cloud (Position Statement)* (Tech. Rep. No. MSR-TR-2010-29). Redmond, WA, USA: Microsoft Research. (http://research.microsoft.com/apps/pubs/?id=121494)

Google. (2012). *Google App Engine Documentation.* (https://developers.google.com/appengine/)

Gotlieb, A., Leconte, M., & Marre, B. (2010, September). Constraint solving on modular integers. In *the CP 2010 Workshop on Constraint Modelling and Reformulation (ModRef 2010).* St Andrews, Scotland: Springer.

Gulavani, B. S., Henzinger, T. A., Kannan, Y., Nori, A. V., & Rajamani, S. K. (2006, November). SYNERGY: a new algorithm for property checking. In *the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2005)* (pp. 117–127). Portland, Oregon, USA: ACM Press.

Hanawa, T., Banzai, T., Koizumi, H., Kanbayashi, R., Imada, T., & Sato, M. (2010, April). Large-scale software testing environment using cloud computing technology for dependable parallel and distributed systems. In *the Third International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2010)* (pp. 428–433). Paris, France: IEEE Computer Society.

Heiser, G., Murray, T. C., & Klein, G. (2012). It's time for trustworthy systems. *IEEE Security & Privacy*, *10*(2), 67–70.

King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, *19*(7), 385–394.

King, T. M., & Ganti, A. S. (2010, April). Migrating autonomic self-testing to the cloud. In *the Third International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2010)* (pp. 438–443). Paris, France: IEEE Computer Society.

Klein, G. (2010, November). From a verified kernel towards verified systems. In *the 8th Asian Symposium on Programming Languages and Systems (APLAS 2010)* (pp. 21–33). Shanghai, China: Springer.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., et al. (2009, October). seL4: formal verification of an OS kernel. In *the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP 2009)* (pp. 207–220). Big Sky, Montana, USA: ACM Press.

Kosmatov, N. (2010a). Chapter 11: Constraint-based techniques for software testing. In F. Meziane & S. Vandera (Eds.), *Artificial intelligence applications for improved software engineering development: New prospects.* Hershey, New York, USA: IGI Global.

Kosmatov, N. (2010b). *Online version of the PathCrawler test generation tool.* (http://pathcrawler-online.com/)

Kosmatov, N., Botella, B., Roger, M., & Williams, N. (2011, March). Online test generation with PathCrawler. Tool demo. (Best tool demo award.). In *the 3rd Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA 2011)* (pp. 316–317). Berlin, Germany: IEEE Computer Society.

Kosmatov, N., Williams, N., Botella, B., Roger, M., & Chebaro, O. (2012, May). A lesson on structural testing with pathcrawler-online.com. In *the 6th International Conference*

on Tests and Proofs (TAP 2012) (pp. 169–175). Prague, Czech Republic: Springer.

Leconte, M., & Berstel, B. (2006, September). Extending a CP solver with congruences as domains for software verification. In *the CP 2006 Workshop on Constraints in Software Testing, Verification and Analysis (CSTVA 2006).* Nantes, France: Springer.

Leroy, X. (2009). Formal verification of a realistic compiler. *Commun. ACM*, *52*(7), 107–115.

Loulergue, F., Gava, F., Kosmatov, N., & Lemerre, M. (2012, July). Towards verified cloud computing environments. In *the 2012 International Conference on High Performance Computing and Simulation (HPCS 2012).* Madrid, Spain: IEEE Computer Society. (To appear)

Marre, B., & Arnould, A. (2000, September). Test sequences generation from Lustre descriptions : GATeL. In *the 15th IEEE International Conference on Automated Software Engineering (ASE'00)* (pp. 229–237). Grenoble, France: IEEE Computer Society.

Marre, B., & Michel, C. (2010). Improving the floating point addition and subtraction constraints. In *the 16th International Conference on Principles and Practice of Constraint Programming (CP 2010)* (Vol. LNCS 6308, pp. 360–367). St. Andrews, Scotland, UK: Springer.

Mell, P., & Grance, T. (2011, September). *The NIST definition of cloud computing* (NIST Special Publication No. 800-145). Gaithersburg, MD, USA: The National Institute of Standards and Technology (NIST).

Michel, C. (2002, January). Exact projection functions for floating point number constraints. In *the 7th International Symposium on Artificial Intelligence and Mathematics (AIMA 2002).* Fort Lauderdale, Florida, USA.

Nipkow, T., Paulson, L. C., & Wenzel, M. (2002). *Isabelle/HOL — a proof assistant for higher-order logic* (Vol. 2283). Springer.

Oriol, M., & Ullah, F. (2010, April). Yeti on the cloud. In *the Third International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2010)* (pp. 434–437). Paris, France: IEEE Computer Society.

Parveen, T., & Tilley, S. R. (2010, April). When to migrate software testing to the cloud? In *the Third International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2010)* (pp. 424–427). Paris, France: IEEE Computer Society.

Riungu, L. M., Taipale, O., & Smolander, K. (2010a, December). Research issues for software testing in the cloud. In *the Second International Conference on Cloud Computing (CloudCom 2010)* (pp. 557–564). Indianapolis, Indiana, USA: IEEE.

Riungu, L. M., Taipale, O., & Smolander, K. (2010b, April). Software testing as an online service: Observations from practice. In *the Third International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2010)* (pp. 418–423). Paris, France: IEEE Computer Society.

Sen, K., Marinov, D., & Agha, G. (2005, September). CUTE: a concolic unit testing engine for C. In *the 5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)* (pp. 263–272). Lisbon, Portugal: ACM Press.

Tillmann, N., & Halleux, J. de. (2008, April). White box test generation for .NET. In *the 2nd International Conference on Tests and Proofs (TAP 2008)* (pp. 133–153). Prato, Italy: Springer.

Williams, N., Marre, B., Mouy, P., & Roger, M. (2005, April). PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In *the 5th European Dependable Computing Conference (EDCC 2005)* (pp. 281–292). Budapest, Hungary: Springer.

Yu, L., Tsai, W.-T., Chen, X., Liu, L., Zhao, Y., Tang, L., et al. (2010, June). Testing as a service over cloud. In *the Fifth IEEE International Symposium on Service-Oriented System Engineering (SOSE 2010)* (pp. 181–188). Nanjing, China: IEEE Computer Society.

Zhang, Q., Cheng, L., & Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *J. Internet Services and Applications*, *1*(1), 7-18.