Making Automated Testing of Cloud Applications an Integral Component of PaaS

Stefan Bucur Johannes Kinder George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

Traditional testing is inadequate for the complexity of modern cloud application software stacks. While the platform-as-a-service (PaaS) model has streamlined application development and deployment, its multiple abstraction layers and dependencies have made testing more difficult. We argue that a modern PaaS offering should include a facility to thoroughly and automatically test a deployed cloud application with only little developer effort.

To support this vision, we propose *layered parameterized tests* (LPTs)—generalized integration tests suitable for cloud applications with multiple processing layers. From LPTs, a testing facility automatically generates concrete tests using *layered symbolic execution*, which focuses on exercising developer-written application logic instead of PaaS library code. We present our design of an automated testing system built on these concepts and demonstrate its use for a modern PaaS.

1 Introduction

Modern consumer software is increasingly relying on the "cloud application" model, where a browser- or mobile device-based client interacts with a functionally rich cloud service. This model is prevalent in major systems like Facebook and GMail, as well as in smartphone and tablet "apps" like Instagram, Siri, or Dropbox. For both developers and users, the economics are highly attractive: cloud-based applications offer ubiquitous access, transparent scaling, and easy deployment at low cost.

The hidden cost is that cloud apps introduce security, privacy, and availability risks. They store and process critical information remotely, so the impact of failures is higher in this model than for single-user desktop apps [17]. This increases the importance of testing for cloud applications.

Rapid advances in development and deployment tools have significantly lowered the barrier to entry for developers, but these tools lack similarly advanced support for testing. Platform-as-a-service (PaaS) offerings, such as Google App Engine or Microsoft Azure, provide easy-to-use interfaces with automated deployment fully integrated into development environments. However, testing tools for apps running on PaaS are still immature, test automation is limited, and developers are left with the laborious and error-prone task of manually writing large numbers of individual test cases.

Integration tests, which complement unit tests by checking that all the components of a fully deployed cloud application work together correctly, are especially tedious to write and set up in such an environment. PaaS-based cloud applications typically use frameworks with a layered communication architecture and perform some processing at each of the layers. Writing a full integration test then requires to carefully craft an HTTP request that successfully passes through all application and framework layers and triggers the desired behavior, while being transformed from one representation to another at each step (e.g., from an HTTP request to JSON to language objects).

Spending precious developer time on testing for improving security and reliability is particularly unattractive in a viciously competitive environment. Promoted by the ability to deploy virtually instantly and at low cost, the pressure is to use features to quickly acquire a large user base. Security and reliability testing is a long-term

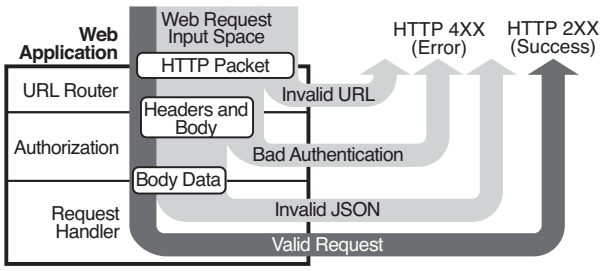

Figure 1: A sample cloud application. Clients communicate with the server through web requests that traverse several layers inside the server before reaching the request handler logic. From the total input space, most possible requests trigger errors in one of the processing layers (light arrows). Only a small fraction of requests successfully traverses all layers and is handled inside the app (dark arrow).

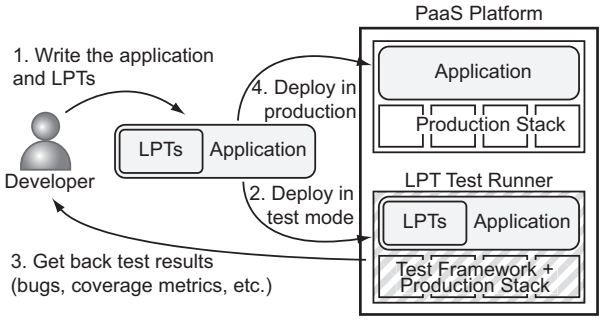

Figure 2: Development flow for using a PaaS-integrated testing service.

investment and does not pay off immediately, thus it is often deferred for later. High-profile failures of cloud applications [16, 9] are thus likely to become a common occurrence, unless we can make testing easy and cheap.

We argue that *testing* must become at least as easy as *deploying* a new app. PaaS has made the latter easy, leaving the former just as hard to do as before. A dedicated testing service must therefore become an integral component of PaaS. Recent commercial test automation systems (CloudBees, Skytap, SOASTA, etc.) relieve developers of managing their own test infrastructure, but still require them to write test suites; we aim to further relieve developers of having to write individual tests. Just as modern PaaS APIs spare developers from having to manage the building blocks of their applications, so should they spare developers from manually writing test cases, and instead offer means to automatically test apps based on only minimal input from developers.

Recent progress in automated test case generation [5, 11, 19] takes an important step in this direction. For a typical PaaS application like the example in Figure 1, a test case generator could use *symbolic execution* (a path-based program analysis) to automatically find HTTP packets that drive the execution to different parts of the code. Unfortunately, these tools are not (yet) a good fit for cloud apps: the number of execution paths successfully crossing all application layers is dwarfed by the vast number of possible error paths (dark vs. light arrows in Figure 1). Yet, most error paths are irrelevant to testing the high-level logic of the app itself (i.e., the innermost layer), because they test code that is part of the PaaS.

We propose *layered parameterized tests* (LPTs) for integration testing of PaaS-based cloud applications (§2). LPTs describe families of integration tests (in the spirit of parameterized unit tests [20]) across several application layers. We rely on developer-provided *onion objects* to describe the layering of the data abstractions in the application; onion objects encode the multiple interpretations of input data as, e.g., an HTTP request, a JSON object, etc. For the automatic generation of thorough test cases from LPTs, we introduce *layered symbolic execution* (LSE), an automated program analysis that is tailored for the layered structure of cloud applications (§3). Finally, we present a design and early prototype for a PaaS-integrated testing service based on LSE (§4).

2 A PaaS Test Interface

We propose an automated testing service integrated in PaaS. Developers write layered parameterized tests (LPTs) and upload them with the cloud application to be executed by the test service. The service uses LPTs to automatically generate application inputs (e.g., web requests and persistent data) that exercise the application layers of interest. The developer writes LPTs by specifying the structure of the application inputs and a target property to be checked.

Developer Workflow Figure 2 illustrates the workflow of a developer using our testing service. The developer writes layered parameterized tests using a platform-provided testing API (step 1). She then deploys the app in test mode, which invokes the LPT test runner of the PaaS (step 2). This test runner is responsible for generating and running the individual test cases from the LPT, and it returns test results back to the developer (step 3). The develop-deploy-test cycle continues until the code is ready to be deployed in production (step 4).

Layered Parameterized Tests An LPT specifies a family of executions (or, equivalently, classes of inputs) plus a set of properties (expressed as assertions) that are expected to hold for all these executions. The specified family of executions can be large or even infinite; still, the test runner can often efficiently check whether the

```
1  from onion import LPT, instrumentLayer
2  from onion import str_, int_, dict_, http_, json_
3
4  app = PhotoUploadApp()
5
6  class TestUpload(LPT):
7      def setUp(self):
8          payload = dict_(
9              [("name", str_()), ("size", int_())],
10             layer="payload")
11         body = json_(payload, layer="body")
12         request = http_(body, layer="request")
13         self.defineInput(request, name="img_upload")
14
15     def runTest(self):
16         request = self.getInput("img_upload")
17         instrumentLayer(request, "request")
18         response = app.sendRequest(request)
19         self.assertNotEqual(response.status, 500)
20
21 class UploadHandler(RequestHandler):
22     def post(self):
23         instrumentLayer(self.request.body, "body")
24         payload = self.decodeJSON(self.request.body)
25         instrumentLayer(payload, "payload")
26
27         # ... process the request ...
```

Onion Object

Figure 3: An LPT example in Python for a photo management application. Highlighted code is the test code written by developers. Names in bold denote the LPT-specific API added to the standard Python unit test framework.

property is guaranteed to hold for the entire family (we give more details on the symbolic execution-based mechanism in §3). A traditional unit test can be seen as a special case of an LPT for which the input is fixed and any provided assertions are checked on a single execution only.

LPTs are defined by developers using a platform-provided API in the implementation language of the application (e.g., Python). The testing API builds on the popular xUnit testing paradigm [3] and extends it with constructs to specify the structure of families of application inputs. The API is easily integrated with existing testing fixtures and frameworks that developers use today.

We illustrate the structure of an LPT and how it is used by the test runner using the example LPT **`TestUpload`** shown in Figure 3, which tests the upload functionality of a photo management application. Under the **`/upload`** URL, the application accepts POST requests containing a JSON-encoded dictionary describing photo information. Figure 4 shows an example HTTP request for the application.

We assume that the app follows the structure given in Figure 1, that it is written in Python using a popular web framework like Django [8] or WebApp [22], and that it is deployed on a PaaS infrastructure like Google App Engine [12] or Heroku [13]. The web framework takes care of dispatching any POST request to the **`/upload`** application URL to the **`post`** method of the **`UploadHandler`** class (lines 21–27). The **`TestUpload`** LPT checks that,

```
POST /upload HTTP/1.1
Host: photo.example.com
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
Content-Type: application/json
Content-Length: 104446

{
  "name": "cat.jpg",
  "size": 104385,
  "data": "gZnJvbSBvdGhlciBhbmltYW[...]"
}
```

Figure 4: An example HTTP request for the upload feature of the photo management application. The text in bold denotes input processed by the application at various layers.

for arbitrary upload requests, the server never returns an internal error (HTTP 500) in its response.

The test runner uses the LPT to automatically generate application input according to the following procedure:

1. The test runner invokes the **`setUp`** method (line 7), which declares the application inputs and their structure as *onion objects* (described below) using the **`defineInput`** call at line 13.

2. Based on the onion objects, the test runner generates a default input for the application.

3. The test runner invokes **`runTest`**, which retrieves the concrete input generated by the test runner with a call to **`getInput`** (line 16). In our example, the input is a web request, which is then sent to the application (line 18). Behind the scenes, the web framework dispatches the request to the **`post`** method of **`UploadHandler`** (line 22). When the handler finishes handling the request, a response object with a status code and body is returned. In our example, the LPT checks that no internal error (HTTP 500) occurred during request handling (line 19).

4. Based on the information collected during the execution of **`runTest`** (described below), the test runner uses the onion objects to generate a new application input (available to the LPT through the **`getInput`** call) and goes back to step 3 for a new iteration. Any assertion failures triggered by the generated inputs are reported to the developers.

Note that the generation and execution of multiple inputs is well suited for parallelization across multiple nodes in the cloud. This allows to leverage the availability of additional nodes to reduce the total testing time.

The test runner uses symbolic execution to generate new inputs (described in more detail in §3). To generate inputs that exercise the application at specific layers, the test runner needs:

- the unwrapped application inputs for the current execution at the different application layers, provided by developers through annotations in the application source code; and

- information about the input structure, provided by the LPT's onion objects.

Annotating Application Layers A web request traverses several processing layers in an application. First, it is received as an HTTP packet string; second, it is decoded into a URL, a set of headers, and request a body; third, the body contents is decoded and processed. Depending on the application framework, processing can involve additional layers, e.g., for converting JSON representations to language objects.

The application layers process data at corresponding layers of the input data (the bold parts of the HTTP request in Figure 4). For instance, the application typically maps the URL to a request handler, checks the headers for authentication information, and processes the body contents in the request handler code.

To expose the application input to the LPT as it is being processed at each layer, developers annotate the variables holding the input data structures in the application source code. Three layers have been declared in Figure 3: the HTTP request at line 17, the request body at line 23, and the JSON payload extracted from the body at line 25. The `instrumentLayer` call attaches a layer name to a variable. Similar to assertion statements, the call is active when executed as part of a test invocation, but disabled in production, where the LPTs are not used. For a typical web stack, only about three layers have to be annotated for each request handler, keeping the required effort on the developer side low.

Onion Objects An *onion object* is a data structure that describes the representations of the application input as it traverses multiple processing layers. The onion object (i) enables more convenient assertion-writing by directly exposing the data layers, and (ii) enables automated test generation to focus on specific layers of the application. Onion objects are needed to specify the application inputs for onion tests, but they can also be used to store output as the cloud application constructs a response in layers.

The framed area in Figure 3 shows the onion object for our running example. The structure consists of a set of *onion nodes* (the identifiers ending in an underscore) connected in a nested structure. There is one onion node for each layer and one for each input structure or value that is supposed to be generated automatically by the test engine. The abstraction level is declared using the `layer` parameter passed to the node constructor and matches one of the layers annotated in the code. Structures and values can be nested within the same layer. For example, the dictionary structure on lines 8–10 has constant keys and wildcard values of type `str_` and `int_`, which mimic the standard string and integer types.

Checking Properties LPTs express application properties through standard xUnit assertion statements (line 19 in the example). Through the dynamic test generation mechanism explained in §3, the test runner actively attempts to generate inputs that cause an assertion to be violated. Each generated test input not failing the assertion serves as a witness for an entire equivalence class of inputs that cannot violate the assertion. When an assertion does fail, the input that caused the failure is reported back to the developer.

To allow input variables at each layer to be used in assertions, each onion node offers a `value` property that refers to the value matched in the current test execution (not shown in the example).

3 Layered Symbolic Execution

In this section, we introduce *layered symbolic execution* (LSE), an LPT execution technique that focuses on covering a particular application layer. LSE uses symbolic execution—a test case generation technique that observes the program structure—to generate inputs in the representation of the layer of interest (e.g., HTTP headers or a JSON payload). Each generated layer-level input is then assembled back into application-level input based on the structure encoded in the onion object, in order to form an integration test case. In the rest of the section, we briefly introduce symbolic execution and describe our technique in more detail.

Dynamic Test Generation LSE is based on dynamic test generation by symbolic execution [11]. Dynamic test generation replaces concrete inputs with symbolic values and executes a single program path symbolically, representing all variables in terms of the symbolic inputs, and building a "path condition" out of all branch conditions along that path (using the negated condition whenever the `else` branch was taken). A satisfying assignment of the path condition is a test input that makes the program follow the path corresponding to the path condition. Negating a constraint in the path condition then yields—if the path condition is still feasible—new test inputs causing execution to diverge from the earlier path. This process can be iterated until all paths are covered. The number of paths can be large or virtually infinite, so the test generation has to settle for covering only a subset of paths. The focus on progressing one path at a time has allowed symbolic execution to be successfully applied to large systems [11, 6, 4].

Naïve application of dynamic test generation to execute the LPT for a cloud app is of little use, however: First, the path exploration can end up exploring many different paths within the framework code, but might test only a single path within the application layer over and over again. Second, the path conditions will encode many branches due to the multiple layers of parsing logic, making symbolic execution of cloud apps prohibitively expensive. Third, if the exploration is unaware of the connections between abstraction layers, blindly negating just single branch conditions will produce many infeasible paths before finding a new valid test input.

LSE and Onion Objects LSE relies on onion objects to mark input variables as symbolic and generate new values based on the alternate path conditions. To this end, each onion object exposes a number of operations:

- `instrument(var)` instruments the variable `var` for symbolic execution, i.e., injects a fresh symbolic input value in dynamic test generation. The variable is expected to match the structure described by the onion object.

- `reconstruct(var, val)` applies an assignment of value `val` to variable `var` that is demanded by the satisfying assignment representing a new path. In doing the assignment, the function performs the necessary modifications to other variables to respect the cross-layer invariants.

- `getDefault()` returns a default value for the object node. It is used for generating the initial test case or any padding values required by invariants (e.g., changing the content length field of an HTTP request requires to extend the actual contents).

For example, applying the `instrument` method of a string onion object on a string variable in Python marks as symbolic the string length and the contents of the character buffer. Then, during symbolic execution, the alternate path constraint yields new values for the length and for the buffer. The `reconstruct` method takes both values and creates a new Python string object.

The reconstruction method is essential for enforcing the object and cross-layer invariants of the input structure. For instance, the length of the reconstructed string would always match the size of its buffer (and avoid spurious overflows); the Content-length HTTP header would always match the size of the new request body, and so on.

LSE Algorithm LSE allows the test runner to focus on exploring paths inside inner application layers. Conceptually, LSE decouples the input layers to give the test runner the flexibility to freely explore an individual layer.

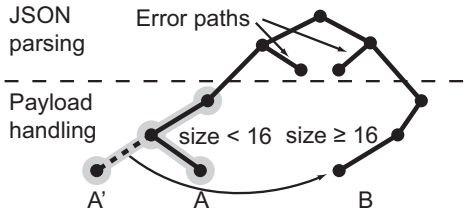

Figure 5: Exploring an execution tree using layered symbolic execution.

When constructing a new application input, LSE reconnects the layers, taking care to respect cross-layer invariants (e.g., the value of a JSON field has to be present also in the HTTP packet). The LSE algorithm proceeds along the following steps:

1. Generate an initial valid input (i.e., a web request) using the `getDefault` call on the root node. The LPT can read this input by calling `getInput`.

2. Symbolically execute the program through the test (the `runTest` method), using symbolic inputs created by calling the `instrument` method on the onion object nodes corresponding to the layer of interest. Any existing symbolic expressions for these variables (which implicitly encode the parsing logic) are overwritten in this step, effectively decoupling the input at the current layer from the previous ones. This permits the symbolic execution engine to negate constraints inside the current layer without being constrained by the previous layers.

3. When the execution completes, negate a constraint in the path condition to obtain new values for the onion nodes.

4. Using the `reconstruct` function of the onion object node, assemble the new values back into a new complete program input (e.g., the HTTP request) for the next iteration.

Figure 5 illustrates an execution tree explored in an iteration of LSE. Consider an initial input for the example in Figure 3, where the value of the `size` field in the JSON request payload is 8 (Path A in the figure). At step 2 of the algorithm, a symbolic value is injected for `size`, together with the rest of the onion object wildcard fields (the highlighted segment of Path A). Now, if the tested path contains the conditional statement `if payload.size < 16`, the `then` branch of the statement is taken and the $size < 16$ constraint is recorded. At the end of the execution (step 3), if this constraint is negated to $size \geq 16$, a new value for `size` is generated, say 20 (the alternate potential Path A'). Then, at step 4, the `reconstruct` functions assembles the new values of all

leaves into a new HTTP packet to be sent to the app, which will cause the `else` branch of the `if` statement to be taken in the next execution (Path B). Note that Path A' is not globally feasible and never explored, but only transiently used to produce the feasible Path B.

Compared to a solution that only marks the variables at the layers of interest as symbolic, LSE is superior in two ways: (1) By obtaining the root input, it is able to run integration tests for a fully deployed application; (2) LSE supports data structures of variable sizes, e.g., arrays whose lengths are symbolic values, by regenerating the input structure at each new iteration.

4 A Testing Platform for the Cloud

We deploy LSE inside a symbolic execution-aware virtual machine (the symbolic VM) that encapsulates the "entire universe" of the application, including the framework and even the language interpreter and operating system, enabling integration testing of the entire application stack. The test-mode deployment then requires just the push of a button for the system to execute the layered parameterized tests, generate coverage statistics, and highlight any failing test cases.

This deployment model leverages the properties of PaaS in several ways: (1) By hiding the testing VMs behind a service interface, the PaaS system can faithfully reproduce the exact environment of production VMs inside the testing VMs without exposing its internals. (2) The testing task can be transparently scaled across multiple VMs by using parallel symbolic execution [4]. (3) Since the application uses standard interfaces for accessing the PaaS components (storage, networking, etc.), the provider is able to substitute production-optimized implementations with testing-optimized stubs that offer a simplified behavior that is better suited to automated program analysis.

From the perspective of the PaaS provider, the test runner service consists of a set of symbolic VMs, operated separately from the production infrastructure. When an application is deployed in test mode, one of the symbolic VMs is allocated for testing: the application code and tests are copied to the guest, and the LSE algorithm is invoked.

Architecture Figure 6 illustrates the architecture of the symbolic VM environment. Inside the VM ①, all application components are symbolically executed in the same low-level representation (e.g., x86 machine code or LLVM [15]). The components execute inside their own vanilla interpreters ②. The test framework ③ plays two roles: it implements (1) the APIs for LPTs and onion objects that developers use to write the testing code, and (2) the LSE algorithm that guides the test case generation.

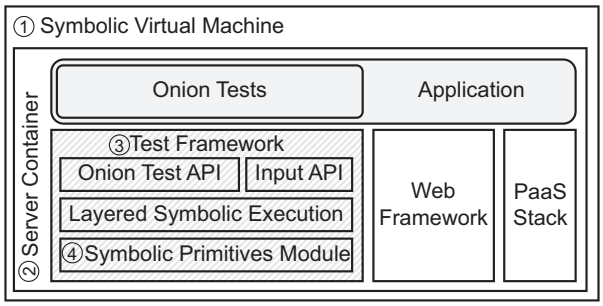

Figure 6: The PaaS test runner service.

Prototype We implemented a prototype of the symbolic VM that tests Python-based Google App Engine PaaS applications and is built on top of the S2E symbolic virtual machine.

In our implementation, the symbolic execution engine and the LSE logic live at different levels in the symbolic VM stack. The symbolic execution engine operates with low-level abstractions such as memory bytes. It resides on the host level, as an S2E plugin that exposes the core symbolic execution primitives to the guest as S2E system calls, e.g., to allow marking memory buffers as symbolic. The LSE algorithm operates on web application state (e.g., by accessing the onion objects), and is implemented in the guest as a native Python extension module. We implemented LSE on top of WebTest [23], a popular fixture library for testing Python web applications. The resulting system is extensible to other languages with limited engineering effort: since the symbolic execution logic is provided at the host level, only the test framework component needs to be implemented in the cloud app language.

Early experiences with the prototype are encouraging: for the full application stack of a simple cloud app, our prototype generates a test case every few seconds.

5 Related Work

Previous work in the space of web application testing on the client or server side focused test case generation on specific types of bugs [18, 1, 2, 14, 21]. On the client, for JavaScript, the Kudzu [18] symbolic execution engine was used to detect code injection vulnerabilities, while Artzi el al. [1] developed a coverage-oriented testing framework to detect programming errors such as HTML validity problems. On the server side, Apollo [2] employs symbolic execution on PHP code to detect runtime and HTML errors, while Ardilla [14] discovers SQL injection and cross-site scripting attacks in PHP. Finally, Wasserman and Su [21] combine static taint checking with string analysis to produce a sound and precise automated checker for SQL injections. Unlike these ap-

proaches, we aim for a flexible platform for automated test generation that is amenable to PaaS integration and allows developers to concisely define the desired properties to check.

Layered parameterized tests are based on the concept of parameterized unit tests [20], which extend test case methods with parameters marked as symbolic inputs during symbolic execution. LPTs further generalize this concept to integration testing across application layers by allowing internal onion object nodes to be marked as symbolic, thus bypassing parsing layers of the program.

Godefroid et al. [10] successfully traverse parsing layers in dynamic test generation by using an input grammar in the symbolic analysis to generate only syntactically valid inputs for a compiler. Onion objects generalize to the multiple processing layers that are typical for cloud applications.

Finally, QuickCheck [7] uses random testing to try and falsify specifications, which share their basic concept with LPTs and parameterized tests in general. LSE leverages the program structure to generate only test cases that cover different paths.

6 Conclusion

We proposed a new PaaS-integrated service for automatic and thorough testing of layered cloud applications. Instead of traditional test cases, developers write *layered parameterized tests* covering a wide range of application behavior, while the service takes care of the concrete test case generation using the novel *layered symbolic execution* algorithm.

Our design decouples the underlying program analysis from the input generation logic and bundles the test generation logic with the rest of the application and its environment. This naturally allows deploying our framework as a service and promises to extend the convenience of PaaS-based development and deployment to testing of cloud applications as well.

Acknowledgments

We thank the anonymous reviewers for their feedback and suggestions for future work. We thank Adam Chlipala, Lorenzo Martignoni, Alexandra Olteanu, and Jonas Wagner for their input on earlier revisions of this paper. This work is supported by the European Research Council and a Google doctoral fellowship to Stefan Bucur.

References

[1] ARTZI, S., DOLBY, J., JENSEN, S. H., MOLLER, A., AND TIP, F. A framework for automated testing of JavaScript web applications. In *Intl. Conf. on Software Engineering* (2011).

[2] ARTZI, S., KIEZUN, A., DOLBY, J., TIP, F., DIG, D., PARADKAR, A., AND ERNST, M. D. Finding bugs in dynamic web applications. In *Intl. Symp. on Software Testing and Analysis* (2008).

[3] BECK, K. Simple Smalltalk testing: With patterns. http://www.xprogramming.com/testfram.htm.

[4] BUCUR, S., URECHE, V., ZAMFIR, C., AND CANDEA, G. Parallel symbolic execution for automated real-world software testing. In *ACM EuroSys European Conf. on Computer Systems* (2011).

[5] CADAR, C., DUNBAR, D., AND ENGLER, D. R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Sys. Design and Implem.* (2008).

[6] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: A platform for in-vivo multi-path analysis of software systems. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2011).

[7] CLAESSEN, K., AND HUGHES, J. QuickCheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN International Conference on Functional Programming* (2000).

[8] The Django project. https://www.djangoproject.com/.

[9] Gmail bug deletes e-mails for 150,000 users. http://www.pcworld.com/article/220886/google_gmail_mail_disappears.html.

[10] GODEFROID, P., KIEZUN, A., AND LEVIN, M. Y. Grammar-based whitebox fuzzing. In *Intl. Conf. on Programming Language Design and Implem.* (2008).

[11] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated whitebox fuzz testing. In *Network and Distributed System Security Symp.* (2008).

[12] Google App Engine. https://developers.google.com/appengine/.

[13] The Heroku cloud application platform. https://www.heroku.com/.

[14] KIEZUN, A., GUO, P. J., JAYARAMAN, K., AND ERNST, M. D. Automatic creation of SQL injection and cross-site scripting attacks. In *Intl. Conf. on Software Engineering* (2009).

[15] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. on Code Generation and Optimization* (2004).

[16] LinkedIn passwords leaked by hackers. http://www.bbc.co.uk/news/technology-18338956.

[17] PERET, S., AND NARASIMHAN, P. Causes of failure in web applications. Tech. Rep. CMU-PDL-05-109, Carnegie Mellon University, 2005.

[18] SAXENA, P., AKHAWE, D., HANNA, S., MAO, F., MCCAMANT, S., AND SONG, D. A symbolic execution framework for JavaScript. In *IEEE Symp. on Security and Privacy* (2010).

[19] TILLMANN, N., AND DE HALLEUX, J. Pex – white box test generation for .NET. *Tests and Proofs* (2008).

[20] TILLMANN, N., AND SCHULTE, W. Parameterized unit tests. In *Symp. on the Foundations of Software Eng.* (2005).

[21] WASSERMANN, G., AND SU, Z. Sound and precise analysis of web applications for injection vulnerabilities. In *Intl. Conf. on Programming Language Design and Implem.* (2007).

[22] The webapp2 Python web framework. http://webapp-improved.appspot.com/.

[23] The WebTest Python testing framework. http://webtest.pythonpaste.org/en/latest/.