Concolic Execution for Django Applications

Luke Anderson, Jon Gjengset, Jeevana Inala, and Andrew Wang
{lukea,jfrg,jinala,wangaj}@mit.edu

1 Motivation

Concolic execution systems allow developers to verify that invariants in their applications are not violated no matter what input is given by a user. This was demonstrated in Lab 3, where the Z3 solver was used to find inputs that would trigger inconsistencies in Zoobar balances. Unfortunately, the lab's framework is written specifically for Zoobar, and would therefore not work for other applications without substantial modifications. Invariant checking is useful for a wide range of applications, and thus we decided to make our 6.858 final project building a generic concolic execution interface for any Django-based web application.

2 Summary of results

Our first step towards running concolic invariant checking on Django applications was to build an application we could test. For this, we decided to port the original lab 3 Zoobar application. By preserving as much of the original application structure and code as possible, while still staying within the confines of idiomatic Django application construction, we were able to test our concolic framework against the same set of invariants used by the Lab 3 version. Specifically, we demonstrate the same functionality as the original application, and show that the same bugs are found in our invariant checker. The code is available at `https://github.com/jonhoo/django-zoobar`.

The next step was to construct a version of the Lab 3 concolic execution framework that could support Django applications. The biggest change we made was undoubtedly to make concolic inputs be preserved throughout the complex internals of Django, so that they reach the application logic intact. This is implemented in `symex/symdjango.py`, which replaces `symex/symflask.py` from Lab 3. In particular, we found that concolic inputs were being lost in the parsing of URLs and POST form data, which is deeply integrated into Django's codebase. Replacing those unsupported operations allowed the concolic values to pass through to Zoobar without losing their symbolic half.

Since real web apps may have large databases, we considered various query strategies with the aim of achieving good coverage while avoiding an exponential increase in the number of paths to check. We implemented three different approaches in `symex/symqueryset.py`: In addition to searching all database items as in Lab 3, we also tried mutating the WHERE clause of SQL queries to capture subtle edge cases, and dynamically inserting concolic values in the database itself. These approaches all have their pros and cons, and in the end, we allow the developer to choose which one to use.

With an eye towards large applications, and with the complexity of Django in mind, we wanted to optimize the constraint-checking framework as much as possible. Several optimizations were added to `symex/fuzzer.py`. Some of these were drawn from *KLEE* (Cadar, Dunbar, and Engler 2008), such as removing implied constraints, and keeping a counter-example cache. The most impactful optimization, though, was to avoid duplicate inputs arising from different path conditions all yielding the same constraints. The concolic checker is available at `https://github.com/jonhoo/django-coex`.

An example of how to initialize the concolic framework, as well as how to issue concolic requests can be seen in `check-symex-zoobar.py`, which was adapted from Lab 3. For reasons explained in the Details section below, the first step is to make explicit the URLs to test, and the database query method to use. The test function and invariants are then defined similarly as in Lab 3, and the concolic tester is invoked at the bottom of the file. Using the default settings of checking all database objects and turning on all optimizations, the framework runs 48 iterations in approximately 30 seconds.

We also ran our concolic execution system on gradapply; the MIT graduate applications website, which is a real application, and substantially more complex than zoobar. Since gradapply is written for an older version of Django (1.6), we first extended our system to support version 1.6. Then it was straightforward to adapt the checking script. The system tests six different URL paths, and completes in 77 iterations. Given the project time constraints, we did not write invariants and find bugs in gradapply, though we believe this should be straightforward now that the checker runs correctly.

Contrary to our initial thoughts, we did not have to modify the Django codebase. This was made possible by the dynamic nature of Python, which allowed us to overwrite or bypass methods that were incompatible with concolic execution during runtime.

3 Implementation details

For each component of the project, we now go into greater detail explaining our design decisions and the challenges we faced.

3.1 Preserving concolic values

1. URL parsing in Django is based on nested regular expressions that are used both to determine which view should be invoked and to do reverse URL lookups. Unfortunately, regular expressions are not supported by Z3, so we needed another way to make the choice of view and the input parameters concolic. The simplest solution we arrived at was to let the developer specify a mapping of views to URLs, allowing them to separately pass in named parameters. This was favored over calling all the views in an application directly, which is more automated, but would also bypass all the Django middleware modules that many applications rely on.

2. Web applications receive input from a number of sources: in-URL parameters, GET/POST parameters, file uploads, cookies, etc., and there is no simple way to automatically determine which of these an application cares about. For the scope of the project, we took the approach of requiring the developer to explicitly construct the datasets with concolic values and pass them in whenever a URL points to a relevant view. Future work could reduce the developer's burden by letting the framework dynamically discover the format of these inputs.

3. So far, the concolic execution framework supports only a fraction of the vast set of Python operations available to Django and web apps. For zoobar, we spent considerable time locating unsupported operations in the Django codebase, such as POST form data serialization and URL resolution, and finding ways to circumvent those operations. It is likely that there are still several corner cases in the Django codebase where concolic values are lost, but the best way to identify them would be additional testing on a diverse set of applications.

3.2 Handling database queries

We aim to achieve good coverage by exposing the internal database query decisions to the concolic checker, while retaining efficiency on large datasets. This goal is difficult to achieve with a single algorithm, as different datasets may or may not expose different branches in applications. The ideal solution is likely a hybrid approach that dynamically adapts to the situation at hand, but for now, we discuss the strengths and weaknesses of the individual database approaches we developed.

1. *All*: This strategy mirrors Lab 3: We iterate over all rows and compare each to the lookup key. We extend this to support lookups with multiple keys, and lookups across related tables. This approach is relatively simple, and generally provides good coverage. However, it potentially re-tests the same

branches, which is wasteful, particularly for large datasets. Further, there are situations (described below in *SQL*) where branches will not be covered.

2. *Mutation*: This strategy is based on *ConSMutate* (Sarkar, et. al. 2012): when a query is made, we extract the WHERE clause, then mutate it by changing the relational operator, e.g. $zoobars > 10$ is one possible mutation of $zoobars \geq 10$. If running the mutated query yields the same result as the original query, we add the symmetric difference of the two conditions as a new path constraint. The hope is that the new constraint will lead to a concrete value that explores an untouched branch of the code. This approach has lower coverage but can potentially catch common programmer errors.

3. *SQL*: This strategy attempts to maximize the branch coverage in applications. To exemplify, consider the two obstacles that may prevent us from exploring both branches of the IF statement below: 1) *users.person.zoobars* is not concolic; 2) there may be no user with > 5 zoobars.

   ```
   user = User.objects.get(username='bob')
   if user.person.zoobars > 5: # do something
   else: # do something else
   ```

   Our solution is to map each database object's properties to new concolic values, e.g., creating a new concolic int for *user.person.zoobars* when *user* is assigned. We also automatically create new database entries when Z3 generates concrete values that are absent, e.g., before we try an input where *user.person.zoobars* is 7, we ensure that there is a database entry with that value. Compared to *All*, we achieve slightly better coverage on Zoobar in even fewer iterations. The downside is that changing the database during a test makes it harder to write invariants.

3.3 Concolic optimizations

Concolic execution invariant checking time is dominated by running the Z3 solver. Runtime can thus be significantly improved by reducing either the overall number of iterations, or by reducing the time it takes to solve each Z3 instance.

1. To reduce the number of iterations, we keep a record of previous inputs to the application, in addition to previous paths, as in Lab 3. Sometimes, different path conditions yield the same solution, in which case the web app would process an identical request. Avoiding this situation reduced zoobar's iterations from 89 to 81. (These numbers do not reflect the database querying strategies being employed to make the concolic framework more efficient.) We suspect the impact of this approach will be more noticeable for larger, and more complex applications.

2. To reduce the complexity of the constraint programs sent to Z3, we detect when some subet of the constraints are implied by others. This is done each time we extend a path condition by a new constraint. Note that determining implication itself is a constraint program, so to avoid running Z3 recursively on these, we determined a set of syntactic rules, which capture most implications in zoobar that Z3 would have found. This further reduced the number of iterations to 75.

3. Sometimes a constraint program may not need to be solved if a solution has already been recorded in a counter-example cache. The cache does not eliminate any iterations, but the amortized runtime benefit of finding a solution in the cache may outweigh the cost of having to run the solver anyway. On zoobar, the counter-example cache was successfully employed 28 times, reducing runtime noticeably simply by reducing the number of costly Z3 solver calls.