Keynote Talk

Under-constrained Execution: Making Automatic Code Destruction Easy and Scalable

Dawson Engler and Daniel Dunbar
Computer Systems Laboratory
Stanford University

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools, Symbolic execution*

General Terms

Reliability, Languages

Keywords

Bug finding, symbolic execution, dynamic analysis.

1. INTRODUCTION

Software testing is well-recognized as a crucial part of the modern software development process. However, manual testing is labor intensive and often fails to produce impressive coverage results. Random testing is easily applied but gets poor coverage on complex code. Recent work has attacked these problems using *symbolic execution* to automatically generate high-coverage test inputs [3, 6, 4, 8, 5, 2].

At a high-level these tools use variations on the following idea. Instead of running code on manually or randomly constructed input, they run it on symbolic input initially allowed to be "anything." They substitute program variables with symbolic values and replaces concrete program operations with ones that manipulate symbolic values. When program execution branches based on a symbolic value the system (conceptually) follows both branches at once, maintaining a set of constraints called the *path condition* which must hold on execution of that path. When a path terminates or hits a bug, a test case can be generated by solving the current path condition to find concrete values. Assuming deterministic code, feeding this concrete input to an uninstrumented version of the checked code will cause it to follow the same path and hit the same bug.

However, these tools (and all dynamic tools) assume you can run the code you want to check in the first place. In the easiest case, testing just runs an entire application. This requires no special work: just compile the program and execute it. However, the exponential number of code paths in a large program makes it difficult to reach all code, even with new symbolic tools. Additionally, code guarded by complex dependencies, for example global configuration options, thread scheduling, or intricate code paths can be tricky to hit in practice. Finally, some programs just cannot run in the test environment and require laborious construction of a fake environment. Operating systems are an obvious example: they do not work when run at user-space, yet almost all checking tools only work on user-space programs.

An alternative is to cut the code to check out of its containing system and test it in isolation. This approach alleviates the burden of executing the entire system and allows testing modules that may not be able to natively run on the testing system. On the other hand, cutting subsystems out of large real systems has proven quite difficult because of their deep entanglement with the surrounding system, colloquially referred to as "environment problem."

Recent work has tried to minimize this effort. JPF introduced *lazy initialization*, which automatically generates complex data structures, potentially guided by user specification of invariants that must hold on them [7]. CUTE [8] does a similar thing using built-in consistency checks. Unfortunately, for large pre-existing systems it is not practical to expect the system to provide built-in consistency checks or to require the user to manually specify input preconditions. DART [6] makes up simple data structure elements *ex nihilo* without user specifications. However, given its blindness to program invariants, what it makes up can easily cause explosions of false positives.

This paper's contribution is the idea of *under-constrained execution*, a simple but powerful twist on symbolic execution that makes it possible to take an arbitrary function and run it without initializing any of it data structures or doing environmental modelling, yet still find quality errors with few false positives. Under-constrained execution lets symbolic values be explicitly marked as being *under-constrained*, indicating that their symbolic values may violate preconditions. (I.e., that constraints on their symbolic values are missing, such as that a pointer is not null.) It then works almost identically to symbolic execution with lazy initialization, but with one change: if an error involves an under-constrained operand u, it only flags the error if it can prove the error must occur for all values of u. Otherwise it asserts that the constraint needed to prevent the error holds and continues execution. For example, given the expression x/u if u is under-constrained but it can prove $u = 0$ (perhaps because of a prior comparison) then it will emit a "divide-by-zero" error. Otherwise it adds the constraint $u \neq 0$ and continues.

ISSTA'07, July 9–12, 2007, London, England, United Kingdom.
Copyright 2007 ACM 978-1-59593-734-6/07/0007 ...$5.00.

Because under-constrained execution explicitly reasons about what it knows for sure (because it has all constraints) and what it only knows approximately (because of missing constraints), it can reason more carefully about symbolic operations. As a result, it can run code whose data is in an invalid state and still get a clean stream of errors out. We tested this claim by using under-constrained execution to run Linux at user-level. We were able call Linux's device driver functions and system calls without doing any setup whatsoever (i.e., not running a single line of init, all data was uninitialized) and yet still find 36 bugs and get many tests that went through over 20 unique branches.

We describe under-constrained execution (§ 2) and then present our preliminary Linux results (§ 3).

2. OVERVIEW

We first discuss how symbolic execution works in our EXE tool [4] and then how to extend this approach to do under-constrained execution.

2.1 Basic symbolic execution in EXE

EXE supports efficient symbolic execution via program instrumentation. It provides a facility for users to explicitly mark which memory locations should be treated as holding *symbolic data* whose values are initially entirely unconstrained. Statements and expressions in the source program are translated in order to operate on such symbolic variables. In the case when the inputs to a particular statement all have exactly one value, i.e. they are *concrete*, the source program is translated in such a way as to directly execute the original code. When the program is executed and a symbolic branch expression is reached the constraint solver is used to determine if the expression is known to be either true or false. If so then the appropriate branch is followed, otherwise the program execution conceptually proceeds non-deterministically along both paths and the appropriate expression is added to the current set of path constraints. In practice EXE uses depth-first search as well as breadth-first and heuristic search to guide the program execution. For further details we refer the reader to [4].

To make this discussion concrete, suppose we use EXE to check a simple network implementation, where read_msg has been instrumented to return symbolic data:

```
read_msg(msg, sizeof msg);
if (msg[8] == 12)
      connection[msg[2]].pkts++;
...
```

After the call to read_msg the contents of msg are symbolic and unconstrained. When we reach the if-statement, EXE will fork execution, on the true path adding the constraint $msg[8] = 12$ and on the false path that $msg[8] \neq 12$. On the true branch, EXE then checks if the value of msg[2] can cause a memory overflow of connection. Since msg comes from the network, it can have any value whatsoever. Thus EXE emits an error. When code reaches the end of a path or error, the current constraints constraints will be solved for concrete values to generate a concrete test of the path.

2.2 Under-constrained execution

Directly applying EXE (or any prior symbolic execution tool) to an isolated component of a larger software system is not as simple as the previous example. Consider the trivial contrived routine in Figure 1 which inserts some value into a global list structure. The function expects that the pointer argument lst is non-null and that the list is unlocked; if the function were simply to be called with symbolic arguments then most of the output would be spurious errors because these preconditions have not been met.

The root of the problem comes from taking symbolic inputs that are missing constraints (because they violate preconditions such as a lock is not held) yet checking them as if all constraints were known by flagging an error if *any* value could cause the error. The basic idea of under-constrained execution is to track which symbolic variables are missing constraints and use this knowledge to more carefully reason about errors.

In under-constrained execution, symbolic values with missing constraints are marked as *under-constrained* to distinguish them from the *exactly-constrained* symbolic variables of the previous subsection (for which we have all constraints). Under-constrained values have the same semantics as exactly-constrained symbolic values except when used in an expression that causes an error to occur. In general it is not possible to know whether or not such an error is real because a missing constraint on the value may make the error infeasible. In this case, we flag an error only if *all* solutions to the currently known constraints on the value cause the error to occur: in this case it does not matter what constraints are missing. [1] Otherwise we add the negation of the error condition as a constraint and continue executing. For example, if an array a of size n is indexed by a under-constrained unsigned variable u then if we can prove u causes an overflow ($u >= n$) we will emit an error, but otherwise will add the constraint $u < n$. Assertions in client code are treated similarly: if EXE can prove an assertion involving an under-constrained value must fail, it emits an error. Otherwise, it adds the constraint that the assertion is true and continues.

Note that program errors involving only concrete values and exactly-constrained symbolic values generate errors as before.

Intuitively, this approach can be viewed as recovering the preconditions from the program's behavior: when we do a check, then in the absence of additional information (i.e., when we cannot prove the condition fails) we assume the check succeeds and add any constraints it assumes to the current set and continue, flagging any subsequent actions that violate the values this constraint forces. This conceptually simple change to traditional symbolic execution transforms it from a technique that is essentially unusable for anything but perfectly set-up code and turns it into a general-purpose, powerful approach that it is possible to shove large amounts of code through.

Testing code using under-constrained execution becomes much simpler; we just specify that the inputs and global data used by a module is under-constrained and start running it. To illustrate this process we step through part of the code in Figure 2, which uses the lock and unlock functions given in Figure 1

Testing of the contrived function in the given code roughly works as follows:

start: Mark lst as under-constrained. If the code had

[1] The exception is that a missing precondition could mean that the path we are on is infeasible; we assume that code is meant to do useful work and explore all paths as long as their branches are internally consistent, similar to what a path-sensitive static analysis would do.

```
 1: elem *list_head(list *lst) {
 2:     elem *e;
 3:     if(lst−>must_lock)
 4:         lock(&lst−>lock);
 5:     e = lst−>head;
 6:     if(lst−>must_lock)
 7:         unlock(&lst−>lock);
 8:     return e;
 9: }
10: void contrived(list *lst) {
11:     elem *e;
12:     if (lst−>must_lock)
13:         lock(&lst−>lock);
14:     e = list_head(lst);
```

Figure 1: Contrived example for illustration

```
void lock(int *l) {           void unlock(int *l) {
    assert(*l == 0);              assert(*l == 1);
    *l = 1;                       *l = 0;
}                             }
```

Figure 2: Simple lock checking code; assumes single-threading.

read global values, EXE would mark these under-constrained as well. Start running the code by calling `contrived(lst)`.

line 12: The pointer `lst` is under-constrained so we assume that `lst→must_lock` is a valid memory reference. At this point our system will create a memory block for `lst` and mark it under-constrained, the details of this are explained more thoroughly below. Since the system has no knowledge about `lst→must_lock` it will assume the branch can go either way, forking execution and asserting its value is non-zero and zero on the true and false paths, respectively.

line 13: One of the processes will call `lock()`. The value `&lst→lock` is under-constrained and thus `lock`'s `assert` will not fire; however after this the value is no longer under-constrained – it is known to be 1.

line 14: Both processes execute a call to `list_head(lst)`.

line 3: At this point `lst→must_lock` is known in one process to be non-zero and in other to be zero; each process will only follow the appropriate path through the branch without forking.

line 4: The process which followed the true path (at line 12) will call `lock()` again. At this point the system will try to prove that the `assert` will fire. In this case it can, since the lock was explicitly set to `1` (by the `lock` call at line 13). The system will emit an error report.

Below we discuss two implementation issues: skipping code and propagating under-constrained values.

Skipping code. Support for under-constrained values allows us to *skip* function calls at will. If a function is skipped then its side-effects are unknown and may have introduced additional constraints on the execution. However, we can model this behavior by marking the function result as under-constrained. Similarly, any local or global data objects that the function might have access to must be (possibly re-) marked as under-constrained. While this may prevent the detection of some errors, skipping functions serves as a useful tool to limit the scope of the analysis without incurring the exponential cost of exploring all reachable paths and could be easily combined with iterative deepening strategy. More generally, we can similarly skip over troublesome constructs, such as certain loops or assembly code, by just marking the locations these constructs may modify as under-constrained.

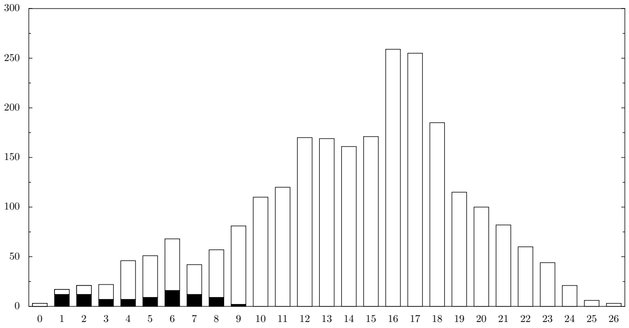

Figure 3: An overlay histogram showing the number of generated test cases which cover the given number of unique branches. This data is from both the base (black bars) and under-constrained (white bars) executions of 15 Linux system calls. Under-constrained execution generated over thirty times as many unique paths (3733 v. 114) and had a significantly better average unique branch depth.

Propagating under-constrained values. For the most part implementing under-constrained execution was a natural extension to EXE. EXE already tracks which variables are symbolic and which are concrete. We simply added the ability to flag certain symbolic values as under-constrained and modified EXE to propagate this flag as under-constrained values are written into new memory locations.

One subtlety arises when a constraint involving both under-constrained and exactly-constrained values is added to the current path condition. In this case it is no longer correct to consider the symbolic value as exactly-constrained. Suppose we execute the statement "`if (s<t) ...`" where `s` is exactly-constrained and `t` is under-constrained. We can no longer treat `s` as exactly-constrained because the unknown preconditions on `t` may restrict its value. Intuitively: since this conditional adds a constraint to `s` in terms of `t`, but `t` is missing constraints, then `s` is now missing constraints as well. The current implementation deals with this problem by propagating the under-constrained flag to exactly-constrained symbolic variables when a constraint involving both is added to the path condition. Unfortunately, this conservative strategy can "taint" exactly-constrained variables unnecessarily. We are currently investigating methods of minimizing this effect.

3. PRELIMINARY RESULTS

The idea of under-constrained execution came from attempting to check Linux device drivers using our initial EXE system. Driver code makes up the bulk of a modern operating system and is notoriously buggy [1, 9]. While drivers ostensibly require a physical version of the device they are intended to drive, they only interact with the device through memory-mapped I/O, which mechanically looks like a memory array with odd read and write semantics. Thus, we can effectively test a driver by marking this array as symbolic and unleashing the driver on it. The driver's resultant interactions with this symbolic memory will drive it down its code paths.

We planned to extract drivers from the rest of the operating system and run them at user level using EXE. Initially, this plan seemed like a mildly arduous adventure with fairly high chances of success. It instead turned out to be completely impractical because of the difficulty setting up the driver environment.

We compiled around 900 drivers from Linux 2.6.19 and defined many stub functions for routines these drivers called but did not define. We also modified around twenty routines to enable deeper checking, such as adding assertions to locking functions and making memory allocation fail. Unfortunately, calling driver routines still requires manually setting up the driver's data structures correctly. While we did so well enough to get the simplest driver functions to work (initialization and cleanup) and find bugs in several dozen drivers, this approach did not work well when we tried to check the more complex routines (e.g., `read`, `write` or `ioctl`). Misunderstandings of subtle dependencies caused almost all drivers to immediately crash after accessing improperly initialized data structures. We spent over six weeks of manual tweaking trying to get environmental initialization right. In the end, while we got a few drivers working, we gave up.

We then had the idea of under-constrained execution. We added a simple implementation of it to EXE and reapplied it to Linux with two changes. First, instead of manually defining stubs as before, we automatically generated implementations that did no work but returned a value marked as under-constrained. (We did reuse the twenty checking functions from the previous effort.) Second, we did no manual driver initialization but instead had EXE automatically mark all driver state as under-constrained. As a result, we could then call any driver code directly, without doing any setup whatsoever.

We then did some preliminary bug finding. We first rechecked the initialization and cleanup code of drivers, getting 36 bugs in total. (We missed some bugs because the base EXE generates a test case that can be rerun using Valgrind, but due to time limitations the under-constrained version does not.) We then ran two of the more complex driver functions `ioctl` and `write`, which had proven previously intractable. We found 4 `ioctl` bugs (out of 146 drivers) and 2 `write` bugs (out of 88 drivers). Most of these bugs could be exploited by malicious user applications.

We also did a crude measurement of how deeply under-constrained execution can push a completely uninitialized complex system. We compiled most of the core Linux kernel and linked it to produce a user-level application. We then called fifteen randomly-selected Linux system calls with symbolic arguments, without doing any setup whatsoever and ran them for five minutes each. We ran this experiment with the base EXE system (no under-constrained execution and no lazy data structure generation) then with the under-constrained version. We measured coverage by counting the unique branches hit on each path (conservatively avoiding double counting for loops, which would otherwise make the under-constrained version look even better). Figure 3 shows these results. As expected, calling system calls without initializing their data structures and without under-constrained execution makes most crash almost immediately. In contrast, the under-constrained version of EXE works very well, and gets extremely far in many system calls despite only running them for five minutes.

4. CONCLUSION

While dynamic tools can check executed paths deeply, they only check paths in code they can run. Empirically, this puts a hard limit on their efficacy. A quick way to see how much they leave on the table is to compare their bug counts to those of a good modern static tool: static tools typically find hundreds to thousands of errors in a large system, while dynamic tool bug counts tend to be in the tens, when they can check a system at all.

Static analysis's secret weapon is that it can check any code it can parse, whether or not is is clear how to dynamically reach that code or run it at all. The goal behind under-constrained execution is to provide dynamic tools with an equivalent ability by giving them a way to take any arbitrary function and run it without doing any setup or environmental modelling whatsoever, yet still find quality errors with few false positives. Our hope is that this approach finally lets dynamic tools supersede static ones in terms of bug counts by making it easy to shove orders of magnitude more code through them.

Acknowledgments. Cristian Cadar wrote the bulk of the base EXE system. Philip Guo set up the device driver framework and found all of the bugs. Peter Pawlowski measured the coverage in Figure 3. This research was supported by National Science Foundation CAREER award CNS-0238570-001, an NSF TRUST center grant, and Department of Homeland Security grant FA8750-05-2-0142.

5. REFERENCES

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys '06: Proceedings of the 2006 EuroSys conference*, pages 73–85, New York, NY, USA, 2006. ACM Press.

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, July 2002.

[3] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software*, August 2005.

[4] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, October-November 2006.

[5] P. Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th Symposium on Principles of Programming Languages (POPL'07)*, Jan. 2007.

[6] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL USA, June 2005. ACM Press.

[7] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

[8] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *In 5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, Sept. 2005.

[9] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *OSDI*, pages 1–16, Dec. 2004.