

Accurate Fault Injection with Concolic Testing

Derrick Coetzee
University of California, Berkeley
Berkeley, CA, USA
dcoetzee@eecs.berkeley.edu

ABSTRACT

Fault injection, modifying the behavior of a program to facilitate the reproduction of hidden or rare problems, is an effective technique for detecting modularity violations in programs, such as missing error handling and implementation-dependent behavior. However, conventional randomized fault injection suffers from poor coverage relative to the number of tests that it runs. In this work, we incorporate techniques from concolic testing to achieve better coverage in client code using a smaller number of tests. Techniques designed to keep false positive rates low are explored, and a number of simple bugs in deployed software are identified.

1. MOTIVATION

In a modular program, the program is decomposed into a set of decoupled reusable modules; each module is given an interface or specification which describes how it is invoked and its expected behavior. In real systems, interfaces include not only signatures such as argument and return types, but human-readable documentation specifying complex semantic properties. A typical example is the `fread` man page, which specifies that “`fread()` and `fwrite()` return the number of items successfully read or written.” The useful abstraction provided by modules is that as long as clients depend only on the interface, any implementation that fulfills that interface correctly can be used interchangeably.

However, in practice it’s easy to write a module that makes assumptions about another module that are not warranted by its interface specification; we call these *modularity violations*. The most common error of this type is forgetting to check a return value for a special value indicating an error; in this case, the (erroneous) assumption is that the function always succeeds. Other examples include:

- Assuming that a hash function always returns distinct values for distinct inputs (is collision-free);

- Assuming that a set data structure will always enumerate its elements in increasing order;
- Assuming that a sorting algorithm is stable (does not reorder equal keys).

Such assumptions are, typically, not documented, and implicit in how the other module is used. Adding to the problem is that many such assumptions are not localized, and cause problems only at a later time due to subtle interactions; for example, if the C allocation function `malloc()` returns NULL, and this pointer is stored in a data structure, the erroneous dereference of that pointer may be deferred to a time long after the allocation.

Modularity violations pose a difficult problem in software quality because the measures designed for conventional (input) testing can easily overlook them. There are two main reasons for this:

1. The assumption may be valid *most of the time* so that exceptions do not arise during testing. This represents a correctness problem with the existing program. The example of assuming a hash function is collision-free falls into this category.
2. The assumption may be valid *for the specific implementation* of the module being invoked, also known as *implementation-dependent behavior*. This represents an implicit coupling between modules that can lead to problems if the implementation of the module being invoked is ever changed. The example of assuming that a set data structure enumerates its elements in increasing order falls into this category; if the set is implemented using a red-black tree, this will tend to occur naturally, whereas if it is implemented using a hash table it will not.

There are two main existing approaches to detecting modularity violations:

1. *Lint-like static code analysis tools*: This white box technique examines the client code and uses static analysis (typically, simple pattern matching) to locate certain suspicious and non-portable constructs in programs and report them. It exhibits both false positives and false negatives. Examples include lint, FxCop, and compiler warnings.

Final project report, CS 265 (Program Analysis, Testing, and Debugging), Fall 2009, University of California, Berkeley. Not a peer-reviewed work. CC0 waiver: To the extent possible under law, the author waives all copyright and related or neighboring rights to this work.

```

void* my_malloc(size_t size) {
    void* result = malloc(size);
    bool simulate_out_of_memory = nondeterministic_bool();
    if (simulate_out_of_memory) {
        free(result); result = NULL;
    }
    return result;
}

```

Figure 1: Fault injection for *malloc()*.

```

ssize_t my_read(int fd, void *buf, size_t count) {
    ssize_t result = read(fd, buf, count);
    ssize_t final_result = (ssize_t)nondeterministic_int();
    final_result = min(result, max(1, final_result));
    lseek(fd, final_result - result, SEEK_CUR);
    return final_result;
}

void f() {
    /* ... */
    ssize_t result = my_read(fd, buf, count);
    if (result == 1234) { abort(); }
    /* ... */
}

```

Figure 2: Fault injection for *read()*.

2. *Fault injection*: This dynamic black box technique instruments the module being invoked and causes it to simulate faults, or more generally, to simulate behavior that fails to satisfy the incorrect assumptions of the client. If the program then exhibits incorrect behavior, it indicates a modularity violation. Typically, fault injection is randomized in order to increase code coverage in the client. As a dynamic method, fault injection may miss some modularity violations, particularly if test coverage is poor. Fault injection may also yield false positives, if it yields a sequence of results that is not feasible according to the specification, or if the specification is incomplete.

In the domain of conventional black box input testing, techniques like random testing and fuzz testing exhibit poor coverage relative to the number of tests generated, since many random inputs cause the program to follow the same code path. *Concolic testing*[11] overcomes this difficulty by observing how input values are used by the program and carefully choosing the next input to force program execution down a different path. Similarly, random fault injection is often unlikely to select exactly the right fault at the right time to cause program failure; it is limited by its lack of visibility into what happens to values after they are returned. In this work, we present a new fault injection solution that takes advantage of concolic testing techniques to achieve better coverage in client code using a smaller number of tests.

In section 2, we (blah blah blah)

2. EXAMPLES

```

map<string, int> hash_history;

int my_hash(string s) {
    if (hash_history.contains(s)) return hash_history[s];
    int result = nondeterministic_int();
    hash_history[s] = result;
    return result;
}

```

Figure 3: Fault injection for a hash function.

Suppose we have a function available that allows a C program to acquire a nondeterministic value of any primitive type at any time. Using this primitive, we can interpose on the interface between two modules and simulate unexpected results. Examples shown in this section use a pseudocode based on C++; the actual implementation is based on C.

In the simplest example, shown in Figure 1, a nondeterministic boolean is used to decide whether or not to simulate an out-of-memory condition. Each time the *my_malloc* function is invoked, another nondeterministic choice is made. All of these faults are *accurate* because the program can, in principle, run out of memory at any time. By running the program many times with different nondeterministic choices and seeing whether the program terminates abnormally, we can validate error-checking for *malloc()* throughout the application.

Simulating a boolean value is simple and does not require the assistance of concolic execution to narrow down the set of useful nondeterministic choices. However, there are other system calls, such as the POSIX *read()* call, that have more complex specifications. According to the *read()* man page, “[i]t is not an error if [the return value] is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because read() was interrupted by a signal.” Figure 2 shows how we might interpose on and simulate unexpected conditions for *read()*. Because the concolic testing engine can trace how the value *final_result* is used even after it is returned to the caller, it can generate choices to reproduce errors such as the “result == 1234” error that are unlikely to be detected by randomized fault injection.

The *read()* example demonstrates enforcement of simple postconditions on the result: for example, the return value must not exceed *count*. If this condition were not enforced, it could lead to *spurious* failures, or false positives, where the caller fails to handle a case that according to the interface specification should not occur. In general, postconditions can be complex, applying not only to the results of a particular call, but to a sequence of calls. For example, suppose we are implementing a hash table data structure which invokes a hash function. The critical property of the hash function that the hash table depends on for correctness is that it is a *pure* function: it consistently maps identical inputs to identical outputs. Other properties, such as the function’s statistical properties, should affect only performance. Collision

handling code in hash tables is often poorly tested because collisions are rare. To enforce the postcondition of pureness, we introduce a *history* map that stores all previously-seen input-output pairs, and returns the same output if the same input is observed again, as shown in Figure 3.

The concept of a history map is a general and powerful one. To explore its full generality, we consider the example of a sort function that invokes a user-supplied comparator function. This comparator C is supplied two elements A and B and determines whether $A \leq B$. To ensure that the sort functions correctly, the comparator must represent a non-strict weak order. That is, for all inputs a, b, c , we must have:

1. Transitivity: $C(a,b) \wedge C(b,c) \Rightarrow C(a,c)$;
2. Totality: $\neg C(a,b) \Rightarrow C(b,a)$. Totality implies reflexivity.

A weak order is used instead of a total order to allow distinct elements to "compare equal."

Figure 4 shows how a history can be used to enforce these conditions. Each time a nondeterministic choice is made, it further constrains the relation, adding a number of new tuples using an ad-hoc but complete resolution procedure. In addition to verifying that our sort implementation is robust against variations in the comparator, this technique can be used to verify that the specification is *tight*: the removal of any of the above three conditions produces an explicit, concrete comparator function that causes the sort to fail:

1. Transitivity: Choose C with $C(0,1)$, $C(1,2)$, $C(2,0)$, $\neg C(0,2)$, and $C(a,a)$ for all a ; since $C(0,1)$ and $C(1,2)$, the sort algorithm may conclude that $(0,1,2)$ is a valid sorted list, but since $\neg C(0,2)$, 0 and 2 are out of order.
2. Totality: Choose the empty relation $\neg C(a,b)$.

Larger-scale examples can be found in componentized applications, where data processing is frequently separated into distinct stages; for example, a compiler may be divided into a front-end parser and a back-end code emitter, or a compression engine may be divided into a front-end transform and a back-end entropy coder. In each of these cases, it is common to *pipeline* the stages, so that back-end stages *pull* output from the front-end stages on-demand; this decreases memory requirements, avoids unnecessary work, and improves cache performance. Such repeated interaction is difficult to model strictly in terms of input and output, but using the fault injection techniques in this paper can be tracked over a sequence of calls. In section 3, we consider the case of IJG's libjpeg, a JPEG compression library which performs a lossy 2D Fourier transform and quantization followed by entropy coding; by simulating arbitrary output from the first phase, we can test the backend coder for dependence on the specific implementation of the transform.

3. EVALUATION

Our implementation is built on the CREST open-source concolic testing tool for C.[1] No changes to the CREST engine

```

map<pair<int, int>, bool> relation;

void add_tuple(int x, int y, bool r) {
    if (relation.contains((x,y))) return;
    relation[(x,y)] = r;
    if (!r) add_tuple(y, x, true); // Totality
    for (((x2,y2), r2) : relation) {
        // Transitivity:
        // C(x,y) ^ C(y,y2) => C(x,y2)
        if (y == x2 && r && r2) add_tuple(x, y2, true);
        // ~C(x,y2) ^ C(x,y) => ~C(y,y2)
        if (x == x2 && r && !r2) add_tuple(y, y2, false);
        // ~C(x,y) ^ C(x,y2) => ~C(y2,y)
        if (x == x2 && !r && r2) add_tuple(y2, y, false);
        // C(x2,x) ^ C(x,y) => C(x2,y)
        if (y2 == x && r && r2) add_tuple(x2, y, true);
        // ~C(x2,y) ^ C(x,y) => ~C(x2,x)
        if (y2 == y && r && !r2) add_tuple(x2, x, false);
        // ~C(x,y) ^ C(x2,y) => ~C(x,x2)
        if (y2 == y && !r && r2) add_tuple(x, x2, 0);
    }
}

bool my_compare(int x, int y) {
    if (x == y) { return true; } // Reflexivity
    if (relation.contains((x,y))) return relation[(x,y)];
    bool r = nondeterministic_bool();
    add_tuple(x, y, r);
    return result;
}

void f() {
    int a[3] = {0, 1, 2};
    sort(a, a+3, my_compare);
    for (i=0; i < 3; i++)
        for (int j=i; j < 3; j++)
            if (!my_compare(a[i], a[j])) abort();
}

```

Figure 4: Fault injection for a sort comparator.

List size	1	2	3	4	5	6	7
Selection sort	1	2	7	33	193	1343	10825
Merge sort	1	2	6	30	145	812	5208
Weak orders	1	3	13	75	541	4683	47293

Figure 5: Number of tests generated for comparator example, compared to total number of weak orders.

were required; instead, all modifications were made to the C source code of the application being tested, and could be included or excluded by conditional compilation. In a more sophisticated implementation, these source modifications could have been fully or partially automated.

We implemented the sort comparator as described in Figure 4 and used it to test both a selection sort and a merge sort algorithm. In each case the number of tests increased rapidly with the size of the list being sorted, as shown in Figure 5, but considerably less rapidly than the total number of weak orders (A000670 in the On-Line Encyclopedia of Integer Sequences [10]); the scheme accomplishes this reduction by focusing on the subset of comparisons actually used by the sort. For both sort algorithms, weakening the postcondition of the comparator introduced errors as expected. Test sets for merge sort may be incomplete, as the concolic testing tool made incorrect predictions sometimes when using this algorithm.

Our next set of experiments was conducted on GNU grep 2.2, an early version of grep (about 34,000 lines of code) prepared for use with CREST by Jacob Burnim. The first interposed on *malloc()* as in Figure 1 to detect missing error handling. 5491 tests were generated, of which 2 indicated actual bugs; 130 tests completed without error despite failed malloc calls thanks to successful recovery mechanisms, and these would have been flagged as false positives by a *lint*-like tool. To decrease the number of tests, we implemented a more sophisticated fault injection technique where *malloc()* would fail again if it had already failed on another memory allocation of the same or smaller size with no intervening free. This limited the number of tests to 2869 and still found the 2 bugs. To further reduce the number of tests, we modified *xmalloc()*, a helper routine which calls *malloc()* and always checks the result, to invoke *malloc()* directly; this dramatically decreased the number of tests to only 75 and still found the 2 bugs.

We instrumented the *read()* calls in grep as described in Figure 2. The number of possible sequences of reads on an n -byte file is 2^{n-1} , the number of compositions of the first n integers. Concolic testing dramatically reduced this to $n+1$ tests. If code is injected into the caller which branches on a specific return value, as in Figure 2, new tests are generated to trigger this branch, as expected. In a separate experiment, *read()* was instrumented to nondeterministically return end-of-file prematurely; this was effective at exploring diverse error-handling paths for end-of-file conditions, and used less tests than varying the input file length itself.

The final experiment with grep involved an optimization of state machine construction in its DFA interpreter. The algorithm uses a hash function to speed up inequality compar-

ison of position sets, which are otherwise slow to compare. We replaced the hash function with a nondeterministic hash function with a history, as in Figure 3. This generated a number of tests that varied from one to thousands depending on the complexity of the search pattern; no defects were detected. Surprisingly, even if the postcondition enforcement on the hash is removed, allowing it to be return different values for the same input, the program still functions correctly; state comparison is apparently only used for coalescing of identical states, again an optimization.

Our last experiment was performed on the Independent JPEG Group’s jpeglib JPEG compression library (213,000 lines of code). [7] This widely-used library for JPEG compression is factored into logical stages: the first performs a lossy Fourier transform and quantization on the image blocks to convert them into the frequency domain, and the second losslessly compresses these coefficients using a simple entropy coding technique. However, to avoid using excessive memory (among other reasons), the compression engine pipelines these stages, working on a line of data at a time. This complex interface makes fault injection a good choice for testing this modular boundary. We discarded whatever results the Fourier transform produced, and replaced them with nondeterministically chosen in-range coefficients.

This was a case where the concolic testing approach was relatively ineffective: because the number of coefficients is very large, even for small images, so is the number of symbolic variables, leading to state explosion in the theorem prover. Additionally, the program performed complex operations on the data that could not be adequately represented using the simple linear arithmetic model of CREST; for example, computing the log base 2 of each coefficient by shifting. For this reason, and because the coefficients were treated symmetrically by the backend coder and limited to 11-bit signed values, random fault injection outperformed concolic testing in this case. In fact, random fault injection succeeded in finding a potential bug, wherein the most negative coefficient value that the backend claimed to support triggered a runtime assertion.

4. RELATED WORK

The ability of CREST and similar concolic testing engines to generate and reason about new symbolic variables dynamically, which forms the foundation for this work’s nondeterministic choice functions, is a little-used feature that works “out-of-the-box.” It appears to be included largely to facilitate convenient introduction of symbolic input at any point in the program.

Fault injection originated primarily as a mechanism for rendering software robust against hardware failure; injection of software faults is still relatively novel. Randomized fault injection tools for software date back to J. Christmansson et al’s work in 1996,[3][4] and have since been heavily commercialized by tools such as Security Innovation’s Holodeck [8] and Codenomicon’s Defensics, [5] which combine interposition on APIs with fuzzing of file and network input. More recent work in this area includes Voas and McGraw’s widely-cited 1997 book on software fault injection, [12] Madeira et al’s evaluation of emulating software faults using SWIFI Xception (2000), [9] and application in new domains such as

fault injection in Java bytecode without source (2008). [6] Traditional randomized fault injection tools as described in these works face two main difficulties: much like random testing, they struggle with the “reachability problem,” where early failures occur with such high probability that they prevent the program from reaching more interesting ones. Additionally, they struggle with accuracy, the problem of ensuring that “injected faults are representative of actual faults.” Unlike the present work, much of the work in software fault injection has focused on undirected perturbation of source code, rather than systematic exploration of behavior consistent with a specification, resulting in poorly-enforced post-conditions.

Other works have briefly considered the idea of using concolic testing techniques to achieve better fault injection. The Enforcer tool of Arthro et al. (2006), [2] designed for injection of I/O failures, considered (but did not develop) guided randomization of the kind used in concolic testing to cope with the reachability problem of automatically fabricating a single test to trigger each program exception.

5. FUTURE WORK

The concolic fault injection process as described herein has a number of frustrating limitations: most noticeably, as in the sort comparator example, complex postconditions can be difficult to express and maintain using histories in imperative code. Ideally, a domain-specific language would be available that would allow postconditions to be expressed in first-order logic; perhaps, if the backend concolic theorem prover supported first-order logic, these rules could be conveyed to it directly in order to achieve better performance and more accurate prediction.

One important application that was not adequately explored in this work is the idea of using concolic techniques to detect portability issues on a single hardware platform; for example, interoperability problems frequently arise when two versions of an application, perhaps running on different machines or with different locales, read and write the same data files. As another example, C programs frequently make assumptions about the layout of data in memory or the byte order of machine words that can lead to unpredictable problems during porting. By varying factors like these nondeterministically, they can be identified on a single machine with less testing resources. These factors are more difficult to test with fault injection, however, since factors like data layout that persist in in-memory data structures are difficult to modify at runtime.

Similarly, library portability issues, in which an application depends on functionality that is only available in a particular version of a library, can lead to difficult-to-predict failures when applications are deployed on machines with different versions of a library - by nondeterministically simulating the behavior of multiple versions, fault injection can help to isolate these problems as well. Conversely, any environment featuring plug-ins such as a web browser must be able to cope with arbitrary communication across the plug-in interface in order to be secure; if this interface is sufficiently rich, this can become difficult, and fault injection can help to anticipate complex failures in this domain as well.

6. REFERENCES

- [1] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. Technical Report UCB/EECS-2008-123, EECS Department, University of California, Berkeley, Sep 2008.
- [2] S. H. C. Arthro, A. Biere. Exhaustive testing of exception handlers with enforcer. In *Formal Methods for Components and Objects (FMCO) 2006 post-proceedings, LNCS 4709*, pages 26–46, Berlin / Heidelberg, 2006. Springer.
- [3] J. Christmansson and R. Chillarege. Generation of an error set that emulates software faults based on field data. In *FTCS '96: Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96)*, page 304, Washington, DC, USA, 1996. IEEE Computer Society.
- [4] J. Christmansson and P. Santhanam. Error injection aimed at fault removal in fault tolerance mechanisms-criteria for error selection using field data on software faults. In *ISSRE '96: Proceedings of the The Seventh International Symposium on Software Reliability Engineering*, page 175, Washington, DC, USA, 1996. IEEE Computer Society.
- [5] Codenomicon. Defensics. Published electronically at <http://www.securityinnovation.com/holodeck/index.shtml>, 2009.
- [6] S. Ghosh and J. L. Kelly. Bytecode fault injection for java software. *J. Syst. Softw.*, 81(11):2034–2043, 2008.
- [7] I. J. Group. jpeglib jpeg compression library. Published electronically at <http://www.ijg.org/>, 2009.
- [8] S. Innovation. Fuzz testing and fault injection with holodeck. Published electronically at <http://www.securityinnovation.com/holodeck/index.shtml>, 2009.
- [9] H. Madeira, D. Costa, and M. Vieira. On the emulation of software faults by software fault injection. *Dependable Systems and Networks, International Conference on*, 0:417, 2000.
- [10] E. N. J. A. Sloane. The on-line encyclopedia of integer sequences. Published electronically at <http://www.research.att.com/njas/sequences/>, 2008. Sequence A000670.
- [11] K. Sen. Concolic testing. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572, New York, NY, USA, 2007. ACM.
- [12] J. M. Voas and G. McGraw. *Software fault injection: inoculating programs against errors*. John Wiley & Sons, Inc., New York, NY, USA, 1997.