Static Analysis and Bugfinding

Alex Kantchelian

09/12/2011

Last week we talked about runtime checking methods: tools for detecting vulnerabilities being exploited in deployment. So far, these tools have been met with a certain dose of skepticism by the industry due to the performance overhead and the fear of compatibility issues. This week we will see what we can do before code is pushed into production.

1 Testing

The first basic approach we can think of is testing the software on *interesting* inputs.

1.1 Fuzzing

Fuzzing refers to a black-box testing technique. In this setting, the source code is unknown so that the only information about the program behavior is acquired by running it with different inputs and inspecting the results. Hence, there are two essential parts in fuzzing: the input generation and the bug detection oracle.

1.1.1 Input generation

The following general techniques can be used to generate test input:

- Manually figure out of good test examples. This is a tedious task and does not in general yield good coverage.
- Completely random input bits. This is almost never used in practice as in general the program will not go deep enough into its path tree to uncover any bugs. Indeed, programs expect in general a pattern in the input, and if not found, then terminate at an early stage.
- Randomly mutate a valid input, by flipping its bits with a small probability p. This is indeed a useful strategy. For example in the case where the input may contain a length field, a random mutation may well cause it to contain a very large value, which the program did not expect.
- Randomly generate an input which would follow the general expected input data template. For example, the JPEG format defines a number of data segments, which all start by byte `0xFF` followed by the segment type identifier byte. The generator would independently generate all the fields and construct a globally valid JPEG file by taking into account the above specification. This approach requires some knowledge of the input file format and takes more efforts to implement than the mutation technique, as in general the format can be much more complex than in the case of JPEG.

1.1.2 Bug detection oracle

Any of the following can be used to detect a bug.

- Crash detector.
- Have several independent implementations of the same program run and compare the outputs. This is in general unpractical, but can be considered in the case the program is expected to output a boolean or some otherwise simple, non-ambiguous structure. For instance, the JAVA bytecode verifier can be scrutinized in this way.
- Manually instrument the code with `assert` instructions. This is a tedious process and the source code is required.
- In general, any of last week's runtime techniques can be used here, for example baggy bounds checking, valgrind's memcheck,... But this would again require access to the source.

Fuzz testing is widely deployed today because it is more or less straightforward to implement, can catch memory corruption errors and has proved to be extremely useful despite its naveness.

1.2 Concolic Testing

Generating inputs that will result in a good coverage of the code's paths is difficult with fuzz testing. Concolic (the contraction of concrete and symbolic) is a possible answer to this issue. The EXE paper (2005) really launches the approach. Concolic testing is a white box technique (source code is available). Instead of specifying the inputs, EXE executes the code symbolically on every expression which depends on the input and normally (concretely) otherwise, trying to go down every possible branch and records the set of constraints the input must verify for each possible execution path. Without considering the optimizations, when an execution path leads to an error (for example, writing to an invalid memory region, a negated `assert`,...), the set of constraints is passed to a solver which either certifies that no input can verify all the constraints, or returns an instance of such an input, thus effectively generating an input of death.

1.2.1 Bug detection oracle

Bounds checking EXE follows a fat-pointer strategy where for each pointer, the base, the length and the concrete value of the pointer are remembered. For example, EXE would detect and exhibit a value for `i` which causes an out-of-bounds read, for example `i=100`.

```
x=malloc(100);
i=read_from_input();
print('%d', *(x+i))
```

At the first line, EXE would remember `{base(x),len(x),x}` and add the following assertion right before the `print` statement:

```
assert(base(x+i)<= x+i && x+i<base(x+i)+len(x+i))
```

Assertions EXE automatically inserts checks in the code. For example, when a division `x/y` occurs, EXE would add an `assert(y!=0)` right before.

In general, symbolic execution allows writing more powerful asserts, essentially in a first order logic form. For example, instead of having the following pointwise check:

```
assert(f(5)!=NULL)
```

EXE would allow more general statements such as:

```
int x=arbitraryInt();
asset(x<0||f(x)!=NULL)
```

1.2.2 Constraints generation and solving

EXE formulates all the constraints on the symbolic variables in the form of a SAT instance which boolean variables encode the symbolic variables' bits. EXE uses a number of strategies to keep this representation manageable by the SAT solver. For example EXE caches the results, and most notably, breaks down large SAT formulas into independent sub problems when this is feasible.

Note that by a special case of Cook-Levin's theorem, every non-deterministic decision Turing machine can be translated to an instance of SAT, meaning that any computable decision function can be encoded by a SAT instance, not just linear decision functions. For example, to encode the constraint `x=y*z` into a SAT instance, one could consider any standard multiplying circuit, name all the wires (in effect create one new variable per wire) and write down the corresponding boolean formula, which would exactly be our SAT instance.

1.2.3 Limits

Aside from not dealing with floating point operations, EXE has a number of important limitations.

Loops EXE can not directly deal with loops that involve symbolic variables for their condition statement. Instead, EXE has to concretize these variables, thus effectively executing the loop a fixed number of times, which is basically equivalent to unrolling it n times, for a fixed n. For example, EXE would not be able to spot the following overflow:

```
char buf[80], *p=src, *q=buf;
while((*q++=*p++)!='\0');
```

Similarly, EXE's lack of proper pointer support lead to poor coverage of code dereferencing symbolic pointers. If `p` is a symbolic pointer, then `**p` would first cause EXE to concretize a possible value for *p. Also, in the example 1.2.1, if `x`'s size was symbolic instead of `100`, then EXE would first concretize it to a possible value (given the active constraints).

Complexity On normal size software, EXE can not get complete path coverage due to the exponential number of many paths. Moreover, SAT is an NP-Complete problem, and although it seems that the average EXE's instances are tractable, there is no guarantee that it will always be so. In particular, a multiplication operation `x=y*z` can be, as of today, extremely expensive to solve, depending on the number of bits. Thus, a lot of interesting work lies in the scheduling algorithm for branch exploration.

Source code EXE requires the source code. However it is theoretically possible to perform symbolic execution on binary as well. The main problem being mapping the small steps sematincs into bigger steps. For example, we would need to recognize a call to malloc.

1.2.4 The subsequent part

Microsoft build SAGE, an EXE-inspired system in 2007. While testing Windows 8, 1/3 of the bugs were found by SAGE and 2/3 by fuzz testing. However, SAGE was only run on 2/3 of the code. One of the bugs that SAGE caught was a switch (inside a while loop) with a number of cases. As it turned out, two 32 bits headers were needed to reproduce the crash, effectively placing this bug out of the reach of random fuzz testing, as in that case, an average of 2^{63} random inputs would have been sampled before crashing. This demonstrates the usefulness of such testing systems.

2 A brief introduction to static analysis

There are several flavors of static analysis. We will talk about abstract analysis. As its name suggests, instead of considering all the possible values that a the program variables can hold, we will abstract these into a well-defined domain, that will be both tractable, and expressive enough.

For example, if `p` and `q` are pointers and we are interested in looking at `NULL` pointer dereferencement, we can define the following abstractions in terms of boolean variables:

$$\tilde{p} = \{p == \text{null}\}$$
$$\tilde{q} = \{q == \text{null}\}$$

Similarly, if `x` is an integer variable and we are looking at signedness errors, we might consider the following boolean abstract variable:

$$\tilde{x} = \{x > 0\}$$

In order to be able to translate each line of the originial code into a line of abstract code, we need to define the abstraction of the usual operations as well. For example, when working with integers signedness as defined above, we can consider the following abstraction for `+`:

$$\begin{aligned} \text{True } \tilde{+} \text{ True} &= \text{True} \\ \text{True } \tilde{+} \text{ False} = \text{False } \tilde{+} \text{ True} &= \top \\ \text{False } \tilde{+} \text{ False} &= \text{False} \\ \top \tilde{+} . = . \tilde{+} \top &= \top \end{aligned}$$

The $\top$ (top) symbol denotes the loss of information on the variable state. It is useful because we still need $\tilde{+}$ to be a total function over the space of abstractions (otherwise we could have only defined $\tilde{+}$ by the first and the third line).

In order to be completely able to run an abstraction of the program, we need one more piece of work: namely how we handle loops. The general idea here is to execute the loop until a fixed point on the abstract values is reached. Such fixed point always exists because of the monotonicity of the abstract operators we have defined. Basically, the abstract values have a lattice structure (e.g. $\text{True} < \top, \text{False} < \top$ in the example above), and all the operators are increasing functions over this structure.

Notice that in the case of finite-state abstract variables, there is only finitely many times a loop can be executed, but this does not need to be so in general and techniques for handling cases where the cardinality of states is infinite exist (see widening/narrowing).

3 Conclusion: Static/Testing/Runtime measures

- Runtime checking cost occurs at runtime by definition.
- Static analysis will cover more all paths. Can find bugs early than runtime.
- Static analysis has a lot of false positives. This is in essence the Rice theorem: we either get a lot of false positive or a lot of false negatives. For today's static analysis tools, 1/3, 1/5, 1/10 are often the proportions of true positives among all the warnings. This is a major cost in terms of time and usability as developers will eventually stop trusting the tool.
- Testing has a zero rate of false positives by definition, but can and does miss bugs.
- Static analysis and testing can't break the system. Much more widely deployed than runtime checks. There is today a significant adoption of static analysis techniques in the industry.