Six ways to automatically find software bugs

FAN Gang, gfan@cse.ust.hk

Abstract

With the rapid development of hardware and the emerging of smart devices like smart phones and wearable gadgets, software, the soul of those devices, becomes more and more important. However, program bugs, which often cause system failure and misbehaviors, is one of the most severe stumbling blocks for even further development of those technologies. Since automatically detecting all bugs in a program is generally undecidable[9], many novel techniques emerged to conquer this program from different aspects and with different tradeoffs. Many of these techniques are commercialized and hence further prove their practical value. In this report, we survey six representative classes of them, and discuss their advantages and limitations.

1 Introduction

Software is a piece of program that describes the intention of how a piece of hardware is going to be used. Well, like everything that made by mankind, programs also contain careless mistakes or misunderstandings, that we call them bugs or defects in programs. Some defects cause system failures, like null pointer dereference, double free, some defects downgrades system performance, like memory leaks, and some defects do not cause failure but rather do not produce the right results. Many approaches emerged to detect and fix them, like testing, manual code review, static analysis, dynamic analysis and so on.

Among them, static analysis stands out with following features:

- Static analysis is more reliable and can be used on time that is not suitable for humans, like midnight.
- Static analysis could detect defects without actually running a program, which means it can analyze incomplete programs(libraries) and it will not downgrade the performance of a program.

- Static analysis can test more program paths than testing.
- Static analysis can be used earlier in the development cycle, it does not need a complete program to start reporting defects.

While static analysis also has a few limitations:

- Some static analyses are hard to scale to large program, they may either use very complex algorithms that could run forever or use large amount of memory space that is not available in most computers.
- Most static analyses have to over-estimate or under-estimate some properties of a program due to many reasons. Some of them may never happen in a real execution. Those over/under-estimations could lead to unreliable defects being reported or real defects being missed.
- Some tool based on static analysis requires manual assistant, which make it less attractive in the industry.

In this report, we will discuss several popular techniques, especially those that are used in commercial tools. Pattern matching described in Section 3 detects bugs by using predefined bug patterns and could be assisted with simple data flow analyses. Detecting defects from program inconsistency described in Section 4 represents a sweet pot in bug detection, tools based on this approach can find some very accurate and persuasive bugs in practice. Detecting bugs by solving graph reachability programs are described in Section 5 and Section 6. Detecting bugs by verifying specifications represents the mainstream of modern bug finding tools, is described in Section 7. Symbolic execution in Section 8 represents a totally different way that simulates real program executions. Some terms are defined in Section 2 and Section 9 summarizes this report.

2 Terminalogy

- **Procedure and Function** Procedure, also know as routines, subroutines and functions, represents a reusable piece of code in a program. It can be used in various parts of a program and at various points of its execution. Procedure and function are used interchangeably in this report.

- **Intra procedural and inter-procedural** Intra-procedural analysis limits its analyzing scope to a procedure, while inter-procedural analysis tracks information across procedure bounds.

- **Control flow graph** A control flow graph (CFG) G(V, E) is a directed graph. A node v ∈ V is called basic block, which contains a sequence of statements that will be executed once the first statement in the sequence is executed. An edge u → v represents the possible flow of control from u to v. The graph contains two distinguished nodes: START has no predecessors and it reaches every node. End has no successors, and it is reachable from every node.

- **Call graph** A call graph is a directed graph that represents calling relationships between functions in a program. Each node in call graph represents a function in the program and each edge (f, g) represents that function f calls function g.

- **Inter-procedural control flow graph** An inter-procedural control flow graph(ICFG) is a combination of the call graph and control flow graphs for a program. It represents the possible control flows over the whole program.

- **Value flow graph** A value flow graph(VFG) is a directed graph. A node in a VFG represents a variable in a program. And a direct edge(u, v) in VFG means whatever value contained in u, it can flow to v at some program point.

3 Find bugs with pattern matching.

Recent research has investigated fairly sophisticated static analysis techniques for finding subtle bugs in programs, the industry experience has been that many interesting bugs arise from simple and common coding mistakes, and can be found using relatively simple analysis techniques. In this section, we are going to talk one implementation Findbugs[16] based on several simple analysis techniques like pattern matching and basic data flow analyses.

First, let us look at an example to get a sense of this technique:

```
1   int intra5(Object x) {
2     if (x == null) {
3       return x.hashCode();
4     }
5     return 0;
6   }
```

This example represents a common mistake made by many programmers. In this example the variable x is checked against null while it is unconditionally dereferenced afterwards. From this snippet of code, we could infer that the programmer's awareness of the variable x could be null and this piece of code should return 0 if it indeed happen. We collect these mistakes and create a bug pattern – "a pointer dereferenced after it is explicitly checked against null and under the true branch of that check." for this mistake.

In Findbugs, all detectors implemented could be roughly divided into four categories based on what kind information they work with:

- *Class structure and inheritance hierarchy only.* Some of the detectors simply look at the structure of the analyzed classes without looking at the code.
- *Linear code scan.* These detectors do a linear scan through the bytecode for the methods of analyzed classes, using the visited instructions to drive a state machine. These detectors do not make use of complete control flow information; however, heuristics (such as identifying the targets of branch instructions) can be effective in approximating control flow.
- *Control sensitive.* These detectors make use of an accurate control flow graph for analyzed methods.
- *Dataflow.* The most complicated detectors use dataflow analysis to take both control and data flow into account.

All detectors are implemented with no sophisticate techniques, the most complex technique is the dataflow analysis. Compare with other approaches we are going to discuss in later sections, the pattern matching approach is considered very lightweight.

3.1 Bug patterns

We describe several bug patterns, some of these patterns are specific to the Java programming language, while others could be extended to detect bugs in other languages too.

3.1.1 Cloneable Not Implemented Correctly

This pattern checks for whether a class implements the Cloneable interface correctly. The most common violation is to not call super.clone(), but rather allocate a new object by invoking a constructor. This means that the class can never be correctly inherited. This detector works on the class structure and inheritance hierarchy only.

3.1.2 Dropped Exception

This detector looks for a try-catch block where the catch block is empty and the exception is slightly discarded. This often indicates a situation where the programmer believes the exception cannot occur. However, if the exception does occur, silently ignoring the exception can create incorrect anomalous behavior that could be very hard to track down.

3.1.3 Suspicious Equals Comparison

This detector uses intra-procedural dataflow analysis to determine when two objects of types known to be incomparable are compared using the equals() method. Such comparisons should always return false, and typically arise because the wrong objects are being compared.

3.1.4 Bad Covariant Definition of Equals

Java classes may override the equals(Object) method to define a predicate for object equality. This method is used by many of the Java runtime library classes; for example, to implement generic containers. Programmers sometimes mistakenly use the type of their class as the type of the parameter to equals(). This covariant version of equals() does not override the version in the Object class, which may lead to unexpected behavior at runtime, especially when the class is used with one of the standard collection classes which expect that the standard equals(Object) method is overridden. This kind of bug is insidious because it looks correct, and in circumstances where the class is accessed through references of the class type (rather than a supertype), it will work correctly. However, the first time it is used in a container, mysterious behavior will result. For these reasons, this type of bug can elude testing and code inspections. Detecting instances of this bug pattern simply involve examining the method signatures of a class and its superclasses.

3.1.5 Equal Objects Must Have Equal Hashcodes

In order for Java objects to be stored in HashMaps and HashSets, they must implement both the equals(Object) and hashCode() methods. Objects which compare as equal must have the same hashcode. Consider a case where a class overrides equals() but not hashCode().

The default implementation of hashCode() in the Object class (which is the ancestor of all Java classes) returns an arbitrary value assigned by the virtual machine. Thus, it is possible for objects of this class to be equal without having the same hashcode. Because these objects would likely hash to different buckets, it would be possible to have two equal objects in the same hash data structure, which violates the semantics of HashMap and HashSet. As with covariant equals, this kind of bug is hard to spot through inspection. There is nothing to see; the mistake lies in what is missing. Because the equals() method is useful independently of hashCode(), it can be difficult for novice Java programmers to understand why they must be defined in a coordinated manner. This illustrates the important role tools can play in educating programmers about subtle language and API semantics issues. Automatically verifying that a given class maintains the invariant that equal objects have equal hashcodes would be very difficult. Findbugs takes an approach of checking for the easy cases, such as:

- Classes which redefine equals(Object) but inherit the default implementation of hashCode()
- Classes which redefine hashCode() but do not redefine equals(Object) Checking for these cases requires simple analysis of method signatures and the class hierarchy.

3.1.6 Null Pointer Dereference, Redundant Comparison to Null

A null pointer is being loaded or an instance field through a null reference results in a NullPointerException at runtime. The detector for this pattern is implemented with a straightforward dataflow analysis. This implementation is strictly intra-procedural, it will not try to decide whether a parameter or a function return value could be null. The comparison against null is taken into consider to make this analysis more precise, for example: for the following code the detector will know that foo is null inside the if body.

```
1   if (foo == null) {
2   ...
3   }
```

If a pointer being compared with null is guaranteed to be non-null, either by a previous dereference statement or a previous comparison with null, the detector will report this statement as a *redundant comparison to null*, which means either this is a useless code or this hints a mistake made by the programmer.

4 Find bugs from program inconsistancy

Detecting bugs from program inconsistency is originally proposed by Dawson Engler[10] and Isil Dilig[8] describe it from a totally different view. Many popular and successful commercial static analyzers like Coverity adopt this technique.

For finding bugs, one of the major obstacles is how to specify what rule should be followed by a piece of program code. One way is to let the programmer specify it, either by writing it to an extra file or by using some special format of comments, which is called *annotations* in many papers. The technique that is going to be discussed in this section – program inconsistency detection – can automatically extract such rule from the source code itself rather than the programmer.

The key to this approach is called program "beliefs". Program "beliefs" are facts implied by code, for example, a dereference of a pointer p implies that p is non-null, a function call $unlock(l)$ implies that the lock l was locked.

For "beliefs", they are separated into two groups, one is "must belief" and the other is "may belief". "Must belief" is extracted from code, and there is no doubt that the programmer holds such belief. A pointer is dereferenced implies that the programmer must believe that the pointer is non-null. "May belief" is the case that we observe the code that suggest a fact, but may instead be a coincidence. A call to function "add" followed by a call to "dec" implies the programmer may believe they must be called pairedly, but it could also just be a coincidence. While if this pairing happens 999 out of 1000 times, then this "may belief" is *highly likely* to be a valid belief.

4.1 Must Beliefs

For "must belief" we look for contradictions, any contradiction implies the existence of an error in the code. The key feature for this approach is that it requires no prior knowledge of the truth: *if two beliefs contradict, we know that one is an error without knowing which one is the correct belief.* For example:

```
1   y = *x;
2   if (x == null){
3     return;
4   }
```

The pointer x is compared with null at line 2 implies that x may be null before this statement, and hence x at line 1 holds the belief that x may be null. While since x is

dereferenced at line 1 implies that the programmer believes that x is non-null, which is a contradiction with the belief extracted from line 2. Then we could conclude that there is an error at this point, either this will cause a null pointer dereference error or the check at line 2 is unnecessary.

4.2 May beliefs

For "may belief" we use statistical analysis to find errors in sets of "may beliefs". Consider a bug checker that finds when a shared variable v is accessed without its associate lock l hold. If we have the priori knowledge of which lock protects which variable, it would be very easy to check this rule using static analysis. Unfortunately, this knowledge does not exist without manual code annotation. However, we can derive this knowledge from the code itself by inspecting what variables are "always" protected by locks. If a variable v is always protected by a lock l, then it may worth attention from developers for where v is not protected by the lock l.

We use an example from [10] to illustrate how a bug could be detected with may beliefs:

```
1   lock i;
2   int a, b;
3   void foo0 {
4     lock(1);
5     a = a + b;
6     unlock(l);
7     b = b + 1;
8   }
9   void bar() {
10    lock(1);
11    a = a + 1;
12    unlock(l);
13  }
14  void baz0 {
15    a = a + I;
16    unlock(i);
17    b = b - 1;
18    a = a / 5;
19  }
```

Consider two variables a and b and the lock l, a is used four times, three times with l held and once without l held. While b is protected by l once and not protected twice. From this observation, we have more confidence for a should be protected by l than b. And this confidence is further strengthened by the fact that a is the only variable that is protected by l at line 11.

Compare with the "must belief" that can give a definitive error with only two contradictory cases, statistic checker based on "may belief" needs more cases to be confident. We are more certain for a belief that happens 999 of 1000 times that a belief that happens 3 of 5 times.

4.3 Summary

Unlike other techniques of bug finding, this approach is trying to find only certain pattern of bugs rather than trying to find all occurrences of bugs or proving the absence of bugs. It can also support some bug types that cannot be or very hard to be detected by traditional techniques like function pairings. Program inconsistency detection in general could act as an augmentation for other techniques we have discussed in this report. The low false positive rate of this approach could improve the accuracy for the whole framework if one adopts it.

5 Finding bugs with the IFDS framework

In data flow analysis, when talk about intra-procedural analysis, a precise analysis gives the results of "meet-over-all-paths" solution. While for a precise inter-procedural analysis, its solution should be equivalent to "meet-over-all-valid-paths" solution. A path is *valid* if the control flow returns to the correct caller function when it reaches the return statement of a function. If an analysis could distinguish valid paths from invalid ones, we call it a context sensitive analysis.

Reps, Horwitz and Sagiv present an inter-procedural data flow analysis framework that focuses a subset but important classes of inter-procedural analysis problems, called the IDFS problem[19]. This class of problems has two key properties: the analysis is distributive and it has only a finite number of possible data flow facts. Many common data flow analysis problems fall into this category, including reaching definitions, available expressions, live variables and possibly uninitialized variables.

We define the inter-procedural, finite, distributive, subset(IDFS) problems as inter-procedural analysis problems with the following properties:

- The set D of dataflow facts is *finite*

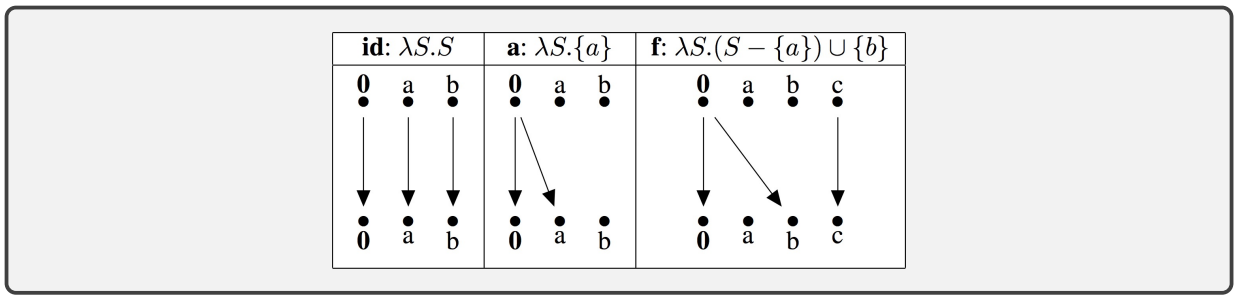

Figure 1: Graph representation for transfer functions

- The flow function f distributes over $\sqcap$
- The domain and range of each flow function is the powerset of the set D (denoted 2^D), the lattice ordering on 2^D is defined as the subset ordering (or, depending on the problem) and the meet operator $\sqcap$ is $\cup$ (or $\cap$, depending on the problem).

With the restrictions imposed by the IFDS framework, then the data flow functions could be represented as graphs. In the following descriptions, we will use subset ordering for the facts domain D and union as the meet operator. Problems that use the superset ordering with intersection as the meet operator can be transformed into their dual problems.

Consider a simple data facts set $S = \{d_1, ...d_n\}$ in 2^D and a distributive dataflow function f. Due to the distributive property of function f, we could have $f(S) = \{f(d_1) \cup ... \cup f(d_n)\}$. It means that for every element in 2^D, the effect of function f could be precisely represented by applying f to each element of D. Since D is finite, there is a very simple graph representation with $2(D+1)$ nodes and at most $(D+1)^2$ edges for f. For example, let $D = \{a, b\}$, then the left side of Figure 1 represents the identity function $\lambda S.S$. 0 is used to represent the *empty* set. Similarly, we could also represent $\lambda S.\{a\}$, the function that adds a to the given set, by the graph in the middle of Figure 1. The function $\lambda S.(S - \{a\}) \cup \{b\}$ for $D = \{a, b, c\}$, which adds b to the given set and remove a from it, is represented by the right side of Figure 1. Further, the graph that represents the composition of two functions could be simply obtained by chaining the two original graphs for the two functions.

We use **super graph** to represent a given program P. The super graph consists of control flow graphs for each function in P. In addition, call edges are added for each callsite to all its possible callees, and return edges are added from the exit node of each function to its all possible callers.

We perform reachability analysis on a graph called *exploded graph*. The nodes in an exploded graph consist of dataflow facts at each program statement. Compare to a super

graph, where we have one node for each statement, there are $(1 + D)$ nodes for each statement, this is also where *exploded* in the name comes from. The edges in an exploded graph form the dataflow functions of a dataflow analysis.

We use a simple example in Figure 2 from [1], which contains a strategy violation if the value returned by $secret()$ could flow to function $printf()$. The exploded graph for this simple program is in Figure 3. In this figure, there are four kinds of edges:

- *Call edges* connect from callsites to all possible callees.
- *Return edges* connect from the return node to all its valid callsites.
- *Call-to-return edges* pass information directly before a callsite to the statements after the callsite.
- *Normal edges* for all other statements

Suppose there is a path in the exploded supergraph from node $n_i k$ to $m_j l$, it means if a data flow fact d_k holds at program point n_i, then the fact d_l will hold at program point m_j. In this way, data flow analysis on supergraph correspond to graph reachability on the super exploded graph. Hence all the dataflow facts that holds at a program point corresponds to all the reachable nodes in super exploded graph from the entry point of the program.

One main problem of this approach is that it is not context sensitive, which means it will generate paths that is inter-procedurally invalid. An inter-procedurally invalid path here means a path that contains mismatched callers and callees. One way to deal with this problem is to annotate labels on the call and return edges. In particular, for each call site s_i in P, we label an associated call edge in $G^{\#}$ with $(_i$ and an associated return edge with $)_i$. Then a valid path is defined as:

$$\texttt{matched} \rightarrow (_i matched)_i matched | \varepsilon$$
$$\texttt{valid} \rightarrow valid(_i matched | matched$$

Based on the above definition, a valid path could be either fully matched caller and with its callees or a matched non-terminal path which represents a valid open path that start from a function and ended in its callees. Hence, we can perform precise inter-procedural dataflow analysis by solving the CFL-reachability[19] problem from the entry node on the super exploded graph. All nodes that are reachable from the entry node along the path whose string of labels is in the context free language generated by above non-terminal *valid*. The time complexity for solving a CFL-reachability problem is $O(N^3 D^3)$.

```
1  void main () {
2   int x = secret();
3   int y = 0;
4   y = foo (x);
5   print (y);
6  }
7  int foo (int p) { return p;}
```

Figure 2: Sample program with information flow violation

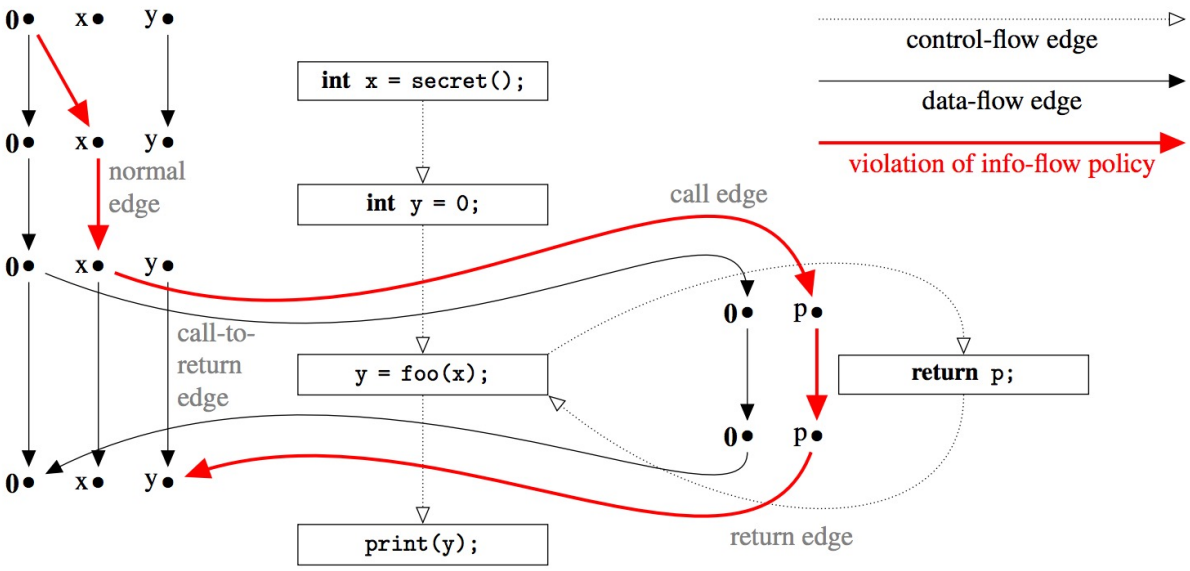

Figure 3: Exploded super graph

The same authors Reps, Horwiz and Sagiv proposed a generalized IFDS: the IDE framework[20]. This generalization of IFDS enables a program analysis to extend the reachability to a value-computation problem. If a dataflow fact $d \in D$ is reachable at a statement, then the IDE algorithm will compute a value from a secondary domain V along all paths that reach d. IFDS can be viewed as a special case of IDE in which its value domain V has only $\{true, false\}$. And the complexity of the IDE algorithm is the same as for IFDS: $O(ED^3)$.

6 Find bugs on a Value Flow Graph

6.1 Source sink problem

Many kinds of bug detection problems could be modeled as source-sink[n,m] problems, it is defined as:

Definition 1. *Consider a program with only three kinds of statements:* ***source*** *statements that produce free values when executed; assignment statements that copy values; and* ***sink*** *statements that received values when executed. A* ***source-sink[n,m]*** *problem is a problem of checking that a dynamic value produced by a source statement will eventually flow into at least n and at most m sink statements in any execution of a the program[5].*

By the above definition, the memory leak detection problem could be defined as a source-sink[1,∞] problem, since we want to make sure that a memory allocated will be freed at least once in any program execution. The double free detection problem could be defined as a source-sink[0, 1] problem, since a memory should not be freed twice. If consider both problems together, the bug detection problem could be defined as a source-sink[1,1] problem, and for clarity, the source-sink[1, 1] problem will be referred as source-sink problem in this report.

6.2 Def use chain and static single assignment form

The Definition Use chain (Def use chain) is a data structure that consists of a definition D, of a variable, and all uses of that variable that can reach the definition without any other intervening definitions. Def use chain is the basis for many compiler optimizations and static analyses.

A counterpart of the def use chain is a use def chain that consists of a use and all its definitions that reach this use without any intervening definitions.

The static single assignment (SSA) form is a form of the intermediate code (IR) that takes the idea of expressing def use/use def relationship explicitly in the code. It achieves this by transforming code into a form that satisfies:

- Each variable defined only once
- Use ϕ-function at control flow joint points

In SSA, each variable is defined exactly once, for a variable that is defined multiple times, versions for this variable are created to represent different definitions. For control flow merging points, if more than one version of a variable is reachable, a special ϕ-function is created to generate a new definition.

6.3 Value flow graph

A value flow graph(VFG) is a graphical representation of a program, the node in a value flow graph are variables in the program and the directional edges in VFG capture the assignment and def-use relationship between variables.

Usually, a VFG is built from a program that is already transformed to SSA format. Built on top of SSA format, the nodes in VFG could be defined as:

- *Global Variable definitions* There is a VFG node for each definition of a global variable.
- *Local Variable definitions* There is a VFG node for each definition of a local variable.
- *Memory regions* There is a VFG node for each memory region.
- *Parameters* There is a VFG node for each formal parameter.
- *Arguments* There is a VFG node for each argument at a call site.

The edges of the VFG include:

- *Assignments* For a statement $x = y$, add an edge $y_n \rightarrow x_m$, n, m are the versions for x and y that reach this statement.
- *ϕ function* For a pseudo function $z_n = \phi(x_m...)$, add an edge $x_m \rightarrow z_n$ for each incoming variable.

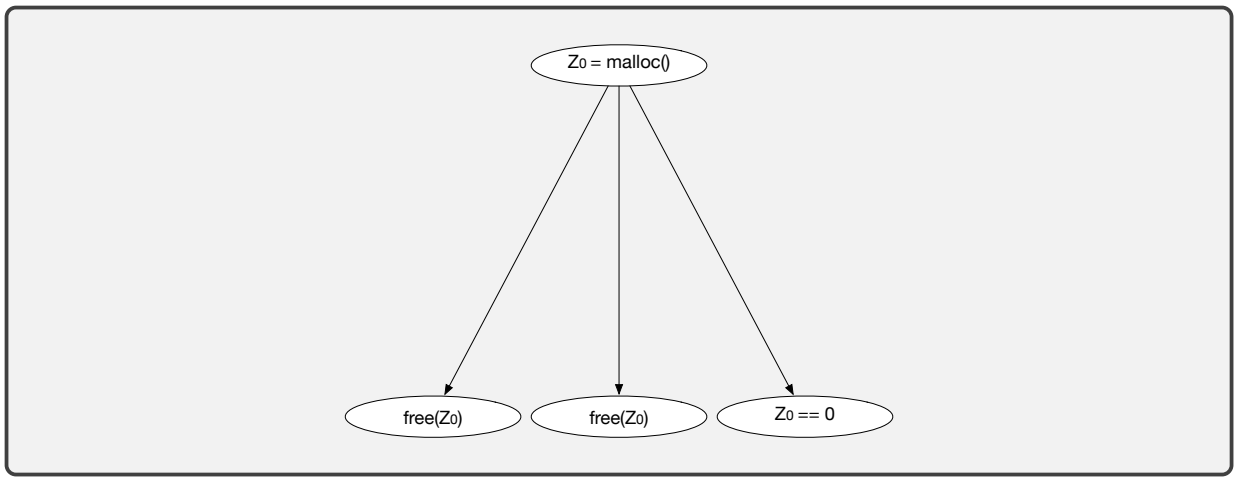

Figure 4: VFG for the sample code

- *Callsite* For a call site $r = f(arg@1, arg@2, ...)$, add edges $v_i \rightarrow arg@i$, $arg@i \rightarrow p_i$, $ret \rightarrow r$, where f is the function, $p_1, p_2, ...$ are the formal parameters for function f, $arg@1, arg@2...$ are the pseudo arguments used by this callsite, $v_1, v_2...$ are the actual variables used by this callsite, r is the variable that receives return value from this function call, ret is the variable that is returned by function f.

For the following sample code, its corresponding VFG is shown in Figure 4.

```
1 z0 = malloc()
2 if (z0 == 0)
3   y0 = 42;
4 else
5   free(z0)
6 output(y2)
7 free(z0)
```

Start from a node in a VFG, we can traverse the graph to find its relevant program points. When doing a source sink reachability on a VFG, only the relevant program points rather than all need to be processed. This kind of analysis is also known as **sparse analysis**. Generally sparse analyses are more efficient than its counterpart – dense analysis, and usually use a worklist algorithm.

6.4 Source sink reachability on VFG

After a VFG has been constructed, for a bug detection problem, we could set all the source and sink nodes. Start from a source node, we traverse the graph to see whether it can reach a sink node and how many can be reached. Some optimization could be used during this

process, like the reachable nodes for a intermediate VFG node could be cached so that later queries on this node could be accelerated. To be inter-procedurally context sensitive, the reachability search on this graph could be modeled as a CFL-reachability[19] problem we have described in Section 5.

One important problem of the VFG is that it produces spurious value flows, aka it generates values flows that cannot happen in any real executions. It is partially because the relationship between different value flows are unknown. For example, if value flows $a \rightarrow b$ $b \rightarrow c$ hold separately on the value flow graph, it does not mean that $a \rightarrow c$ holds in a real execution. It may due to $a \rightarrow b$ and $b \rightarrow c$ cannot happen on one real path, or it may because the value of b from a is killed when $b \rightarrow c$ happens.

Cherem and et.al addressed this problem with a novel approach[22, 5, 21] that adds **guards** to the edges of a value flow graph. A guard annotated on a value flow graph edge represents the control flow condition that for a value flow. For an edge (a, b) in a value flow graph G, there exist a statement s corresponds this value flow, aka the value flow from a to b when statement s executes. A statement is successfully executed implies that its previous branch conditions are satisfied. We annotated this condition on the value flow graph edge to express that when this value flow happens, the annotated conditions must be satisfied. After a unconditional graph reachability search, we will have a bunch of traces that we call them candidates. For each trace, we collect the guards along the value flow trace edges, and solve it using a SAT solver. If the solver answers no, it means there is no such path in a program that can satisfy this trace, and hence implies this trace is invalid.

7 Find bugs by verifying specifications

Finding a bug could be viewed as finding violations of a specification that a program should satisfy. For a null pointer dereference problem, we have a specification that a pointer being dereferenced should not be null. For a memory leak detection problem, we have a specification that a memory allocated should be freed along every valid program execution path. From this point of view, a series of on-demand static analyses are proposed.

Weakest precondition (WP) is such a technique to infer the conditions backwardly from the interest program point to either an entry of a function or an entry of the whole program. Let us assume we want to verify a program where we know the postcondition R but not the precondition[18]:

$\{?\}S\{R\}$

$\{P\}$
$\{P_0\}$
$s_1;$
$\{P_1\}$
$s_2;$
...
$s_{n-1};$
$\{P_{n-1}\}$
s_n
$\{R\}$

Figure 5: Program S and its pre- post-conditions

In general there could be many or even infinitely many preconditions Q work on program S that can satisfy the postcondition R. However, there is precisely one precondition that describes the necessary set of conditions such that the execution of S will lead to a state satisfying R. This Q is called the *weakest precondition*. A condition P_1 is weaker than P_2 if $P_2 \rightarrow P_1$.

For an example program S: $y = x * x$ and a postcondition R:$y \geq 4$, the weakest precondition Q will be $(x \leq -2) \cup (x \geq 2)$

Assume we have a program $s_1; s_2; ...; s_n$ with precondition P and postcondition R. We want to check whether this postcondition could be satisfied with given precondition. Starting from s_n and R, we can produce P_{n-1} and some constraints that must be satisfied to successfully execute s_n. And the results are so-called *verification conditions*. P_{n-1} is now the weakest precondition for statement s_n and the postcondition for statement s_{n-1}. We repeat this process until P_0 is generated. P_0 accumulates all the conditions along the program, that is $P_0 \rightarrow R$. The remaining of this problem is to prove $P \Rightarrow P_0$, if we succeed to do so, then we can be sure that the program $s_1; s_2; ...; s_n$ satisfy this specification (Or a bug can be triggered).

This approach based on weakest precondition computation is also known as *backward symbolic analysis*. In general, a backward symbolic analysis computes weakest preconditions over each control flow path, backwardly from an interesting statement to the entry point. If the precondition for any path is satisfiable, then the assignments for it is *guaranteed* to guide the execution from the entry point to the interesting statement.

Backward symbolic analysis is generally *demand driven*, which means a such analysis will only explore statements that is relevant rather than all. And due to its demand driven nature, backward symbolic analysis could also be applied to incomplete programs, e.g.

libraries.

A representative tool Extended Static Checking for Java (ESC/Java)[15] is a tool that is powered by verification condition generation and automatic theorem proving techniques. It provides programmers with a simple annotation language with which programmer design decisions can be expressed formally. ESC/Java generates the preconditions from code and examine it with the design decisions recorded in the annotations. It warns inconsistency and potential runtime errors if found any.

Here is an example of annotation before a definition of a function in ESC/Java:

```
//@ requires input != null
```

The @-sign at the start of this Java comment tells ESC/Java that this is an ESC/Java annotation and it tells the checker that this function has a precondition of *input != null*.

And to specify the design decision that a field is always non-null, the user can annotate the declaration of that variable with:

```
/*@non_null*/ int[] elements;
```

Similarly, the non null annotation could also be used in a function's formal parameter:

```
Bag(/*non_null*/ int[] input);
```

Another annotation *invariant* is used to specify a property that the programmer intends to hold at every routine boundary for a variable:

```
//@ invariant 0<= size \&\& size <= element.length.
```

The annotation could be used as a precondition that is assumed to hold at the beginning of a function, and could also be used as an assertion that must be held when being called.

When used as a precondition, we infer the verification guards from interesting program points and compare whether the annotated precondition is inconsistent with the inferred precondition. For example a pointer dereference site s that dereference pointer p, if we have annotated that p is non null, then we will not report s as a potential null pointer dereference defect.

And when used as assertions, we will check whether we could find a path that violates this assertion. For example, warnings will be raised if there is any program place that assigns null to field *elements*.

A much earlier work LCLint[11, 12, 13] for c language shares the same spirit with ESC/Java. Due to their nature of limiting analysis scope within a function, they have these advantages:

- *Flexible* Writing new checkers with these tools are generally quite easy, LCLint and ESC/Java both provide a language to easily write new checkers.

- *Incremental* The results for a function is based on the specification written and the function itself. There is no strong tie between functions. So most of them could be implemented to support incremental feature by only analyzing the functions that changed.

- *Scale* This kind of analysis is easy to support large projects. This is a common advantage of intra-procedural analyses.

ESC/Java and LCLint are intra-procedural analyses, that is the analysis itself will not go across function boundaries. So the user annotation is needed to specify preconditions that should hold when a function is called in the program. This approach requires additional manual effort from programmers, which makes it less attractive. To reduce the this limitation, the ESC/Java team has developed Houdini[14] to infer specifications for unannotated programs. ESC/Java and LCLint also suffer from the out of synchronization program, that is the code will be out of synchronization with the annotation, in this situation the annotation will not express the genuine programmer design decisions any more and could even express wrong design decisions.

The backward symbolic analysis also suffers from the path explosion problem, that is the number of feasible paths need to be explored grows exponentially with an increase in program size, and could even be infinite if a program contains unbounded loops. A tool named Marple[17] addresses this issue by categorizing program paths as they relate to the targeted statement. Marple is used to detect buffer overflow vulnerability in a program, and categorizes paths to $infeasible, safe, vulnerable, overflow-input-independent$ and $don'tknow$. This categorization enables the checker to prioritize the paths to be searched for finding a root cause of a buffer overflow vulnerability.

Infeasible A infeasible path means a path that cannot be executed. Infeasible paths will produce false positive reports or create spurious value flows in a static analysis. Previous report has shown that 9-40% of the paths in the program can be identified as infeasible paths[2].

Safe Some paths to execute a potential buffer flow statement are guaranteed to be safe regardless of the input.

Overflow-Input-Independent Not all buffer overflows are exploitable, that is some buffer overflows are independent from input.

Don't Know Detection of some buffer overflow paths are beyond the ability of static analysis. Like paths that involve complex pointer aliasing, complex arithmetic calculation, library calls and system environment.

Modern program consists of many sub-routines, aka functions or procedures. A static analysis that can analysis data flows across function boundaries is called an inter-procedural analysis. Marple supports inter-procedural analysis by applying its backward symbolic analysis on top of ICFG rather than CFG.

ICFG is a combination of CFG and Call Graph(CG), it represents the "MAY" control flows of the whole program rather than a single function.

ESP[6] proposed by Manuvir Das et al. and Snugglebug[4] proposed by Satish Chandra et al. take another approach – function summary. Let's assume we are processing a function foo, and we encounter a call to function bar. We want to apply a transfer from the call site in foo to the return node of bar. The problem here is we cannot summarize the effect of function bar before analyzing function foo since there is external information that is unavailable when analyzing bar. Like the actual arguments that are passed to bar and program states when bar is called. To solve this problem, we create a summary to summarize the function's behavior, while leave the inputs and external information as symbols of that summary. When bar is called at a callsite in foo, we instantiate an instance of bar's summary with the information provided at the callsite, and a full transfer function is created.

8 Find bugs with symbolic execution

Symbolic execution[3] is a technique of analyzing and simulating a real execution of a program. During the simulation, inputs to the program are represented with symbols that can be any value. When the simulation engine meets an expression, it will evaluate this expression in terms of the symbols and variables in the program. When the simulation encounters a branch expression, it will take conceptually both branches and the constraints generated as outcomes of selecting a branch.

We use an example to illustrate using symbolic execution to find a bug, this program will fail if the $read()$ function returns 6:

```
1    y = read()
2    y = 2 * y
3    if (y == 12)
4      fail()
5    print("OK")
```

When a symbolic execution engine is used to simulate the program execution, it does not have the concrete number for the return value of $read()$. So a symbol 's' is assigned to it. Then 's' is also copied to y after the execution of $y = read()$. The statement $y = 2 * y$ assigns "2 * s" to y. The statement at line 3 has two control flows: the true branch if y's value is 12 and the false branch otherwise. The engine then creates a constraint "2 * y == 12" when true branch is taken, which means a real execution reaches the fail statement if '2 * s == 12' is true. Then the engine resolves this constraint with a constraint solver, which will give it an answer of s = 6, which is the necessary condition that makes this program fail.

One of the advantages symbolic execution has over other static analysis techniques is that whenever it finds a bug, the answer from the constraint solver will automatically form a test case that will trigger this bug. This feature will help developers to quickly reproduce this bug and more heavy runtime debugging techniques could be involved to better understand this bug.

Traditional program testing focus on "code coverage", while in reality a bug will be triggered only when certain execution path is taken. And a statement is covered by one test case does not mean it is tested under all possible conditions. Symbolic execution has the advantage of exploring all feasible paths of a program, even those that are very hard to trigger with manual written test cases.

A tool called KLEE[3], which its underlying technique is symbolic execution, manages to find 10 fatal bugs in well programed and tested project $COREUTILS$ (e.g., ls, is there any program well tested than it?), these bugs had escaped detection for 15 years.

With above many advantages, symbolic execution has two main limitations, one is scalability and the other is interaction with the external environment.

Symbolic execution simulates all feasible program paths, while the number of all paths in a program grows exponentially with the size of the program. It could even grow to infinite when the program contains unbounded loop iterations and recursive function calls. This problem could be partially solved by using heuristics to guide the path exploration process

to achieve higher path coverage, and to unroll the loops certain number of times. Since exploration of different paths are independent in nature, multiple paths could be explored simultaneously to leverage the multi-core processing power of modern computer.

Symbolic execution treats any program inputs as symbols, without special assistance, it cannot be aware of the relationship between inputs. One simple example is the main function in c language, it has two formal parameters *int argc*, which represents the number of options, and *char** argv*, which contains the actual option strings. Without external assistance, symbolic execution will assign two totally unrelated symbols to these two arguments, which will lead to infeasible program paths and further lead to spurious bug reports.

9 Summary

In this report, we discussed six popular static analysis techniques when they are used for finding bugs. Some of them are very efficient like pattern matching, while some of them are very heavy in terms of computing complexity and scalability, like symbolic execution and full path sensitive backward symbolic analysis. After conducting this survey, we could end it with these three conclusions for the future of designing a good tool for finding bugs:

First, to reliably detect bugs, the underlying technique should be at least path sensitive. Path insensitive analysis tends to generate many spurious reports, while when we are talking about the application for bug finding, generating less spurious reports is at least as important as finding more bugs.

Second, SMT solvers are used in almost every technique. Symbolic execution uses it the check whether adding one more step will make the path invalid. Tools based on backward symbolic analysis use it to generate the weakest precondition or find unsatisfiable conditions accumulated. Even simpler techniques like pattern matching and annotation based approaches could use it as an infeasible path pruner. The satisfiability solver could certainly be used in many other aspects of bug finding. Discovering new uses of these solvers could be a future research topic. Moreover, with more reliable solvers like Z3[7] being opened to public, it becomes more possible for these techniques to be applied to the industry.

Third, there are still some low hang fruits that should not be ignored by the tool designer. Like the two techniques we have discussed – pattern matching and program inconsistency. Tool designers tend to use more sophisticated techniques and try to cover all bug cases in one round. I would suggest integrating these simple but accurate information into their

tools. They could provide additional evidence for already found bugs and can also find new bugs that are hard to detect by heavy techniques.

10 References

[1] Bodden, E. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis* (New York, NY, USA, 2012), SOAP '12, ACM, pp. 3–8.

[2] Bodík, R., Gupta, R., and Soffa, M. L. Refining data flow information using infeasible paths. *SIGSOFT Softw. Eng. Notes 22*, 6 (Nov. 1997), 361–377.

[3] Cadar, C., Dunbar, D., and Engler, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 209–224.

[4] Chandra, S., Fink, S. J., and Sridharan, M. Snugglebug: A powerful approach to weakest preconditions. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 363–374.

[5] Cherem, S., Princehouse, L., and Rugina, R. Practical memory leak detection using guarded value-flow analysis. *SIGPLAN Not. 42*, 6 (June 2007), 480–491.

[6] Das, M., Lerner, S., and Seigle, M. Esp: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2002), PLDI '02, ACM, pp. 57–68.

[7] De Moura, L., and Bjørner, N. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008), TACAS'08/ETAPS'08, Springer-Verlag, pp. 337–340.

[8] Dillig, I., Dillig, T., and Aiken, A. Static error detection using semantic inconsistency inference. *SIGPLAN Not. 42*, 6 (June 2007), 435–445.

[9] D'Silva, V., D'Silva, V., Kroening, D., and Weissenbacher, G. A survey of automated techniques for formal software verification. *TRANSACTIONS ON CAD* (2008).

[10] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. Bugs as deviant behavior: A general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev. 35*, 5 (Oct. 2001), 57–72.

[11] Evans, D. Static detection of dynamic memory errors. *SIGPLAN Not. 31*, 5 (May 1996), 44–53.

[12] Evans, D., and Evans, D. Using specifications to check source code.

[13] Evans, D., Guttag, J., Horning, J., and Tan, Y. M. Lclint: A tool for using specifications to check code. In *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering* (New York, NY, USA, 1994), SIGSOFT '94, ACM, pp. 87–96.

[14] Flanagan, C., and Leino, K. R. M. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity* (London, UK, UK, 2001), FME '01, Springer-Verlag, pp. 500–517.

[15] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2002), PLDI '02, ACM, pp. 234–245.

[16] Hovemeyer, D., and Pugh, W. Finding bugs is easy. *SIGPLAN Not. 39*, 12 (Dec. 2004), 92–106.

[17] Le, W., Le, W., and Soffa, M. L. Marple: A demand-driven path-sensitive buffer overflow detector.

[18] Popov, N., and Popov, N. Verification using weakest precondition strategy. in computer aided verification of information systems – a practical industry oriented approach. *PROCEEDINGS OF THE WORKSHOP CAVIS'03, 12 TH OF FEBRUARY, E-AUSTRIA INSTITUTE* (2003).

[19] Reps, T., Horwitz, S., and Sagiv, M. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1995), POPL '95, ACM, pp. 49–61.

[20] Sagiv, M., Reps, T., and Horwitz, S. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci. 167*, 1-2 (Oct. 1996), 131–170.

[21] Steffen, B., Steffen, B., Knoop, J., and Rüthing, O. The value flow graph: A program representation for optimal program transformations. *PROCEEDINGS OF THE EUROPEAN SYMPOSIUM ON PROGRAMMING, PAGES 389–405. SPRINGER-VERLAG LNCS 432 432* (1990), 389–405.

[22] Sui, Y., Ye, D., and Xue, J. Detecting memory leaks statically with full-sparse value-flow analysis. *Software Engineering, IEEE Transactions on 40*, 2 (Feb 2014), 107–122.