Static Vulnerability Analysis

Static Vulnerability Detection helps in finding vulnerabilities in code that can be extracted by malicious input. There are different static analysis tools for different kinds of bugs, like format string attack, buffer overflow, and other such vulnerabilities. Following block diagram shows the placement of static analyzers, the input to them and expected output.

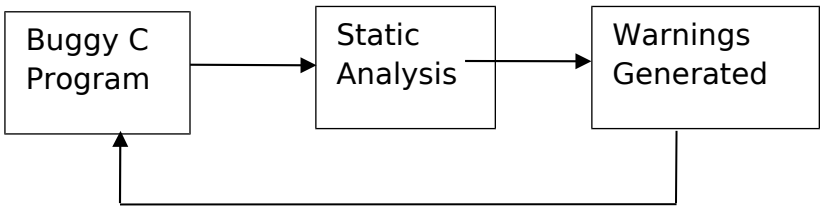

Fig.1 Block diagram of generalized static analysis tool.

- Static analysis has an advantage over runtime analysis. Since runtime analysis requires checks that reduce the performance of the program. However when the same changes are done in the code itself, it is not considered an overhead!

Let us first introduce few terms:

1. **Flow insensitive:** A technique which does not consider the order of execution of statements while analyzing code is called flow insensitive.
2. **Path insensitive:** A technique which considers all possible paths, and not just the semantically possible paths in the code is called path insensitive. Here path means control flow in the code.
3. **False Positives:** A False Positive is when you think you have a specific vulnerability in your program but in fact you don't.
4. **False Negatives:** A False Negative, as the name suggests is opposite to False Positives. Here the vulnerability is hidden and not identified.
5. **Complete:** A technique which gives no false positives is called complete.
6. **Sound:** A technique which gives no False Negatives is called sound. So here there might be lots of false positive warnings but we can rest assure that all the vulnerabilities will be flagged.
7. **Monomorphic:** A monomorphic analysis does not consider the call site of a function. It concatenates all calls to a function and generalizes the resultant graph. This approach might create many false positives, but it is easier to implement.
8. **Polymorphic:** A polymorphic analysis considers the call site of a function and formulates different results for different calls of the same function. Thus giving a more precise but also more expensive analysis.

Consider reading the following paper for more information on them. Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C by Jeffrey S. Foster, Manuel Fähndrich, Alexander Aiken.

There will be two types of static analysis tools that will be explained in the lecture.

1. Tools that use Data Flow graphs.
2. Tools that use Control Flow Automata (Similar to control flow graphs, except for code on edges)

Data Flow Graphs:

Consider static analysis of format string attacks for studying data flow graphs:

```
Ex 1: Line 1:    void log_use (char *username) {
      Line 2:        char buf[512];
      Line 3:        strcpy (buf, "Logging from: ");
      Line 4:        strncat (buf, username, 400); // Taken 400 bytes for avoiding buffer overflow
      Line 5:        printf (buf);
      Line 6:    }
```

The above code has a format string vulnerability. Let us see how we detect it using static analysis. We introduce two keywords **$tainted** and **$untainted** in the compiler**.** These will be assigned to variables depending on the source of the value in the variable. All variables that get their values from untrusted source will be tainted. Now any untrusted source will eventually have to do a *read()* of the value. (We are not considering recv and other variants that can receive untrusted values over the network)

So let us redefine the header containing declarations of read() and printf()

*int read (int fd, $tainted char *buf, int len);*

*int printf ($untainted const char *fmt, …..);*

Here we have added the required keyword to the parameters depending on what the function assumes its parameters to be. However, if a $tainted data type is passed when a $untainted data type is expected, the compiler will give a type mismatch warning, but the code is not broken. Which means we can actually pass $tainted data when a function expects $untainted data. Hence, $untainted data becomes subclass of $tainted data. (Just like vector is a subclass of objects in Java)

$untainted char ∈ $tainted char

$untainted int ∈ $tainted int

Where ∈ means subclass. Let us now formulate a data flow graph for the following piece of code. Assume that data read using *read* is passed in as the username argument to log_use, so we can treat this argument as tainted.

```
Ex 2:  Line 1:    void log_use (char *username) {
       Line 2:        char *p = username;
       Line 3:        char buf[512];
       Line 4:        strcpy (buf, "Logging from: ");
       Line 5:        strncat (buf, p, 400); // Taken 400 bytes for avoiding buffer overflow
       Line 7:        printf (buf);
       Line 8:    }
```

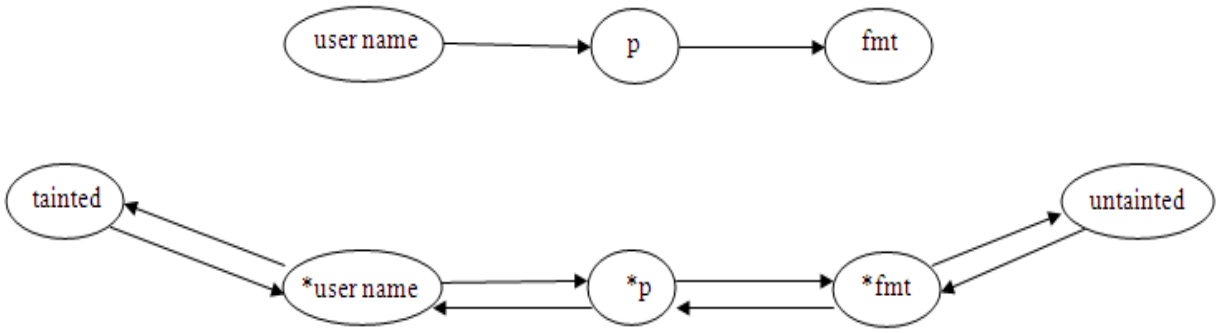

Fig.2 Data Flow Graph for code in example 2.

The above graph is formulated on the basis of how data flows from variables.

There are two graphs above, the first graph with no double edges shows the data assignment. The second graph with tainted and untainted nodes is more useful for analysis. Here is an explanation on why there is a double edge in graph 2 for a single edge in graph 1.

Consider the following line of code:

*p = username;

Here *p and *username are the same value. So no matter which pointer changed the value of variable, they will both be affected. Therefore, for all pointers, there will be double edge (going to and coming from) with the pointer it shares the address of variable.

In the above graph there is a path from tainted to untainted. This static analysis gives a warning and thus recognizes a format string vulnerability.

Consider the same example with following modifications:

```
Ex 3:   Line 1:     void log_use (char *username) {
        Line 2:         char *p;
        Line 3:         char buf[512];
        Line 4:         strcpy (buf, "Logging from: ");
        Line 5:         strncat (buf, username, 400); // Taken 400 bytes for avoiding buffer overflow
        Line 7:         printf (p);
        Line 8:         p = username;
        Line 9:     }
```

Here, the purpose of program has changed, however we are more interested in concluding something else. Note that this new piece of code will give the same data flow graph, but the vulnerability is no longer there. This is an example of Flow-Insensitive analysis. So the code flow is not considered while formulating or traversing the graph. Hence this analysis can give false positives.

Let us consider ex 1. And form a data flow graph for it. Here src and dst come from the definition of strncat.
*int strncat (char *dst, const char *src, int len);*

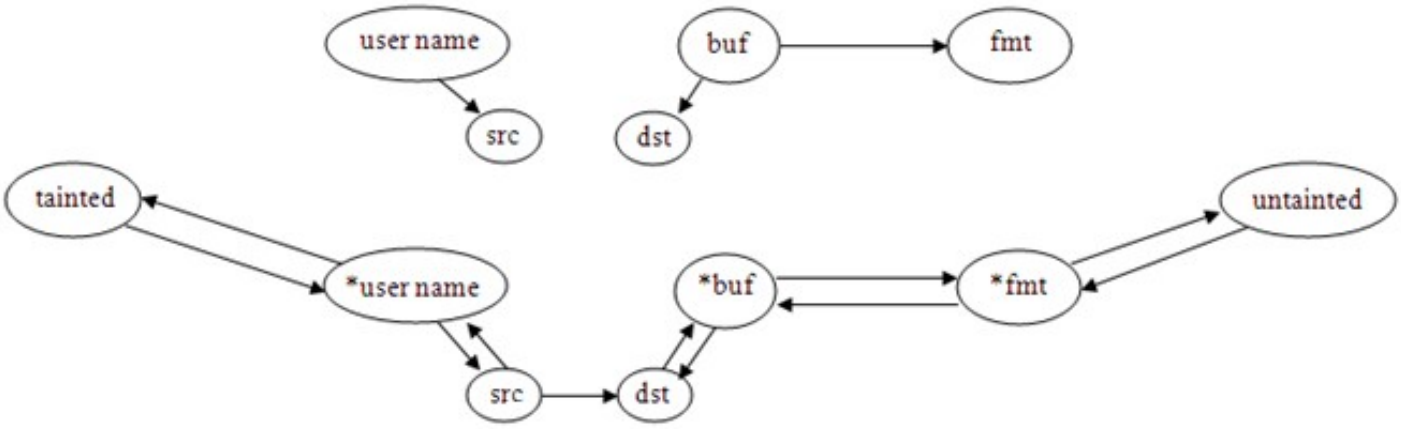

Fig.3 Data Flow Graph for code in example 1.

This graph like the previous gives a data flow graph. This graph also has a path from tainted to untainted data. Hence, this accurately detects a vulnerability. Now consider an extension to this graph. Suppose there is a call to the function *strncat()* from somewhere else in the same program. Let us consider how this graph changes.

strncat (b, a, …);

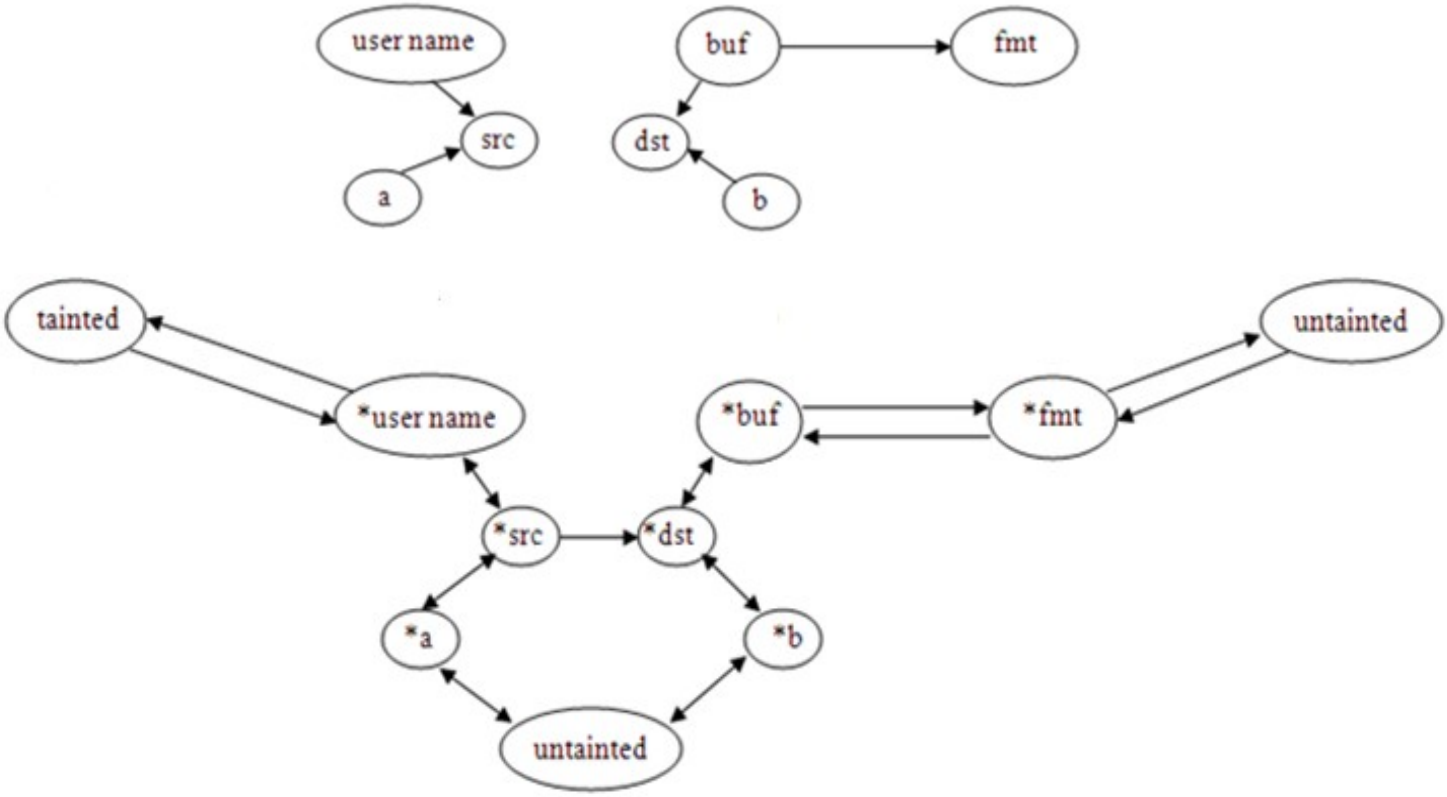

Fig.4 Data Flow Graph for code in example 1 and considering another call to strncat.

Here we assume that *strncat(b, a, n)* is not a vulnerability. That is both the source and destination strings are untainted. However, in graph traversal there can be a path as follows:

*a → *src → *dst → *buf → *fmt → untainted.

This path is not semantically right. Hence all calls to strncat will give false positive warnings. This graph is an example of monomorphic analysis. That is the call site of strncat was not considered while traversal of graph. To make this analysis polymorphic we need to form a context free language which will be associated with the site of function call.

Consider there will be $(_1$ and $)_1$ as the identifiers for data flowing in and out of 1st instance of strncat. So on so forth. Using this approach, the graph would now look like follows:

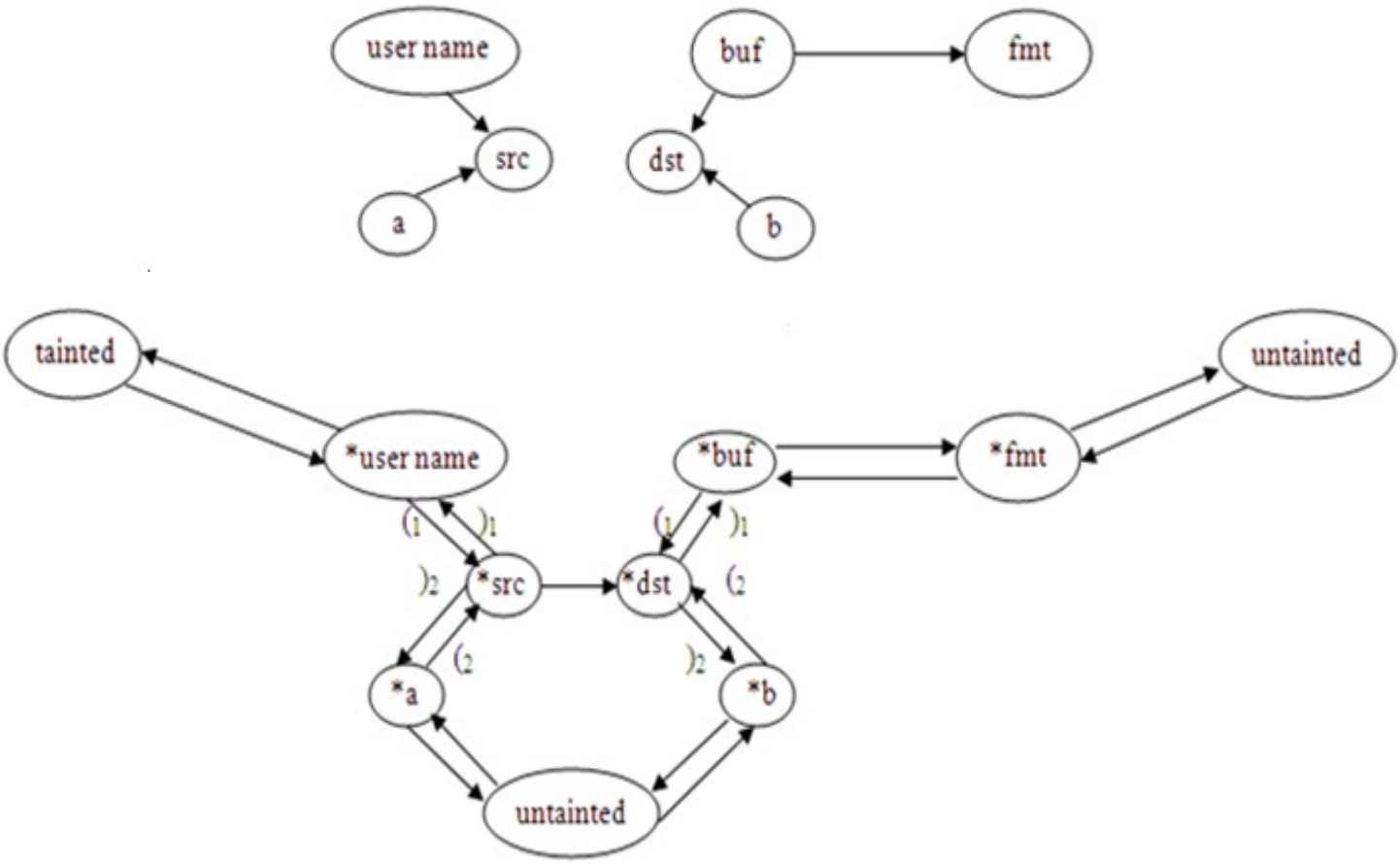

Fig.5 Data Flow Graph for code in example 1 using polymorphic analysis.

Here while traversing the graph we take note that if the function was entered by a certain identifier, the path taken to leave the function should be of the pair of this identifier. For example, if function is entered from an edge with identifier $(_1$, then the path with identifier $)_1$ should only be considered. This approach gives a sound analysis of the code, which covers all possible vulnerabilities and has no false negatives. CQual and Qink are examples of tools for static analysis of C.

Control Flow Automata:Control Flow Automata is important for analyzing control flow bugs. That is, when we need to find if there are vulnerabilities which will cause the control of code to flow in a direction not designed to be. Control flow automata is different from control flow graph in the sense that it uses code statements on edges instead of nodes.

Consider the following piece of code:

Ex 4:

```
Line 1:        if (….)
Line 2:           setuid (0);
Line 3:        if (….)
Line 4:           system (…);
Line 5:        if (….)
Line 6:           setuid (≠0);
Line 7:     }
```

Here there will be a control flow bug if the *if* condition 1 is satisfied and then control flows to *system(…)* after doing a *setuid(0)*. Let us see how to detect such vulnerabilities statically using form control flow graphs.

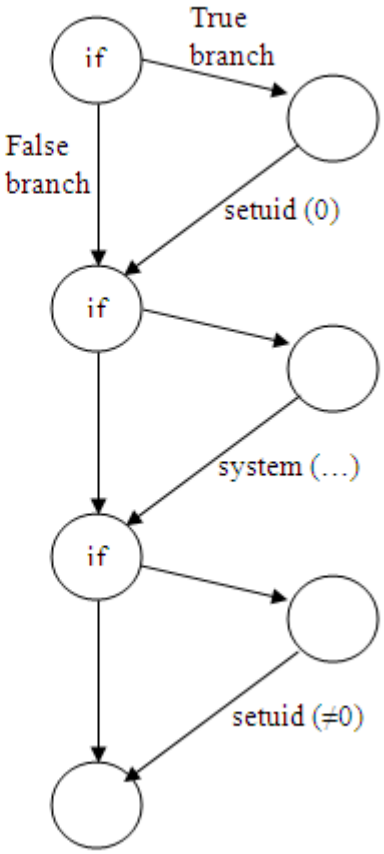

Fig.5 Control Flow Automata for code in example 4.

Let this grammar be called C and its Language L(C) = Superset of all the control flows the original program would have. That is in this graph we consider paths that might not be possible semantically in the code, we are doing this by not considering the condition in *if*. Such an analysis is called path-insensitive. Now, consider the following input to DFA of another grammar M.

$\sum$ = {setuid (0), system (…), setuid (≠0)}

Here L(M) = Vulnerability found.

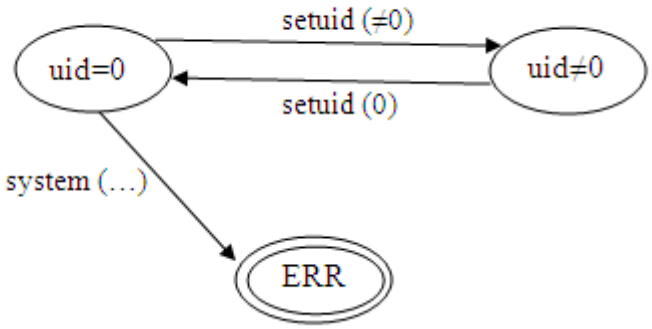

Fig.6 Directed Finite Automata for code in example 4.

In this DFA, all other edges are self edges. Which does not change the state and hence for simplicity not shown.

Now, we have to combine the two state machines and form a single state machine. In the new language if there is a path to the ERR state, then the analysis has found a vulnerability.

In the above two languages, if L(C) ∩ L(M) is empty, then we have no vulnerabilities. However, this analysis is not free from False Positives, since a superset of actual paths in code is considered for analysis. This might show vulnerabilities that might not exist due to the semantics of the code.