Chapter 36
Static Analysis Tools Concepts

Revision 1.0, November 2025.

Objectives

- Learn a little history of static analysis tools
- Understand how static analysis tools work
- Understand the limitations of these tools
- Understand the role of the analyst when using these tools
- Learn about control and data flow graphs used by some tool

36.1 Background

In this chapter, we will learn about the important topic of software scanning tools, also known as static analysis tools or SAST (Static Application Security Testing). Using these tools is a key part of any programmer's workflow. In this chapter, we will give you some background on code analysis. In the next chapter, we will show you how to use these tools to improve the quality and security of your code.

You might ask why you should be interested in how the tools work when you have no intention of being a tool builder; just want to use them. By understanding how they work and their limitations, you will be able to better understand their output and be a much more effective tool user.

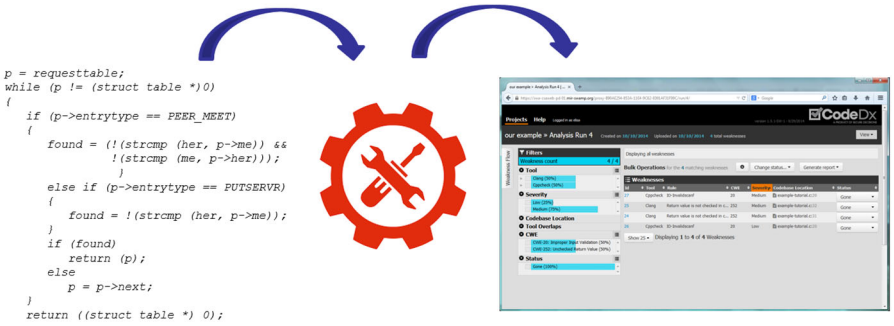

Figure 1: Work Flow for Using Static Analysis Tools

When you use a static analysis tool, you start with the program – in source code, byte code, or binary code form – and scan the program with the tool to get a report on what problems were found in the program.

Note that these tools are static, which means that we do not run the program; these tools just look at the code in source or binary form to understand what the program might do.

36.2 Start with the Compiler

Your first line of defense is the compiler. You might not have viewed the compiler as your ally in producing better code, but it has a lot of capabilities in this area. In the process of analyzing your code, the compiler can detect many potential problems. Compilers use warnings for problems that are not in violation of the language standard but indicate risky behavior.

For example, there used to be some interesting vulnerabilities based on dangerous use of a C or C++ printf format string[1]. Having too many format items could allow a program to print the contents of its stack and using the `%n` format element could allow writes to arbitrary locations in memory.

Compilers now check for weaknesses associated with dangerous use of format strings. In the GNU and clang C and C++ compilers, `-Wformat` option causes the compiler to check that the types of the elements in a `printf` format string match the listed variables and that dangerous elements like `%n` are not used.

You should turn on as many of these warning as you can handle. They will help ensure that your program behaves as you intended. You might think that in gcc or clang, `-Wall` is really all the warnings. It now includes quite a few, but not all the possible warnings provided by the compiler as programmers get upset when new options are added to this list. As of the date that this chapter was written, `-Wall` includes 81 specific warnings[2]. The gcc (and clang) compiler introduced the `-Wextra` option to make the list more complete. As of this date, this includes 29 additional warning. You use `-Wall` and `-Wextra` together, but there are still more individual warnings that you can specify (though these warnings get pretty esoteric).

If you want to be really careful, you can also use the `-pedantic` option to make sure that you get warnings that strictly adhere to the ISO language standards.

The Microsoft Visual Studio compiler makes life a bit simpler here: `/Wall` means all the warnings.

36.3 Lint: The First Static Analysis Tool

At Bell Labs in 1970, Ken Thompson and Dennis Ritchie introduced the UNIX operating system, a groundbreaking and important innovation. The original version of UNIX was written in assembly language. In the 1973, after much discussion, Dennis Ritchie introduced the C programming

[1] `https://en.wikipedia.org/wiki/Uncontrolled_format_string`
[2] `https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html`

language and Version 4 UNIX was then rewritten in C. You should understand that C was quite an advance from the typical assembly language approach taken by previous operating system teams.

The original C compiler was pretty primitive in the kinds of things that it checked in the code, so in 1978 at Bell Labs, Stephen Johnson introduced lint, a software tool that did a variety of interesting and innovative checks on the C code.

Along with a basic analysis of the program's control and dataflow, it could check for things like variables being used before they were initialized and reporting if a calculation might overflow.

The invention of lint spurred the compiler writers to improve their code analyses, so as lint got smarter over the years, so did the compilers. This is an interesting synergy between compilers and static analysis tools that continues today.

Note that all static analysis tools that you find today can be considered descendants of Johnson's lint.

36.4 What Can Static Analysis Tools Do?

Static analysis tools are designed to find flaws in the code or, as we defined the term in Chapter 2, find weaknesses. The weaknesses may have come from simple mistakes or perhaps were placed in the code intentionally.

Ideally, you should run these tools from the very start of your software project. However, even if you have a mature project, you should use these tools.

> Before any commit to your software repository, static analysis tools should be run and the weaknesses addressed.

It is important to note that these tools do not know what the program is intended to do or how it does it. Instead, they look for specific programming errors and violations of good programming practices. The tools are is structured like a compiler, but instead of producing a binary as output, they produce diagnostic messages.

Let us talk now about the capabilities of these tools. Ideally, these tools are accurate in that they do not miss important weaknesses (*false negatives*) and do not report too many non-existent weaknesses (*false positives*). You want the tools to be fast. If they are too slow, programmers simply will not use them. You also want them to check lots of different things, from memory errors to injections to races, and many more. The unfortunate reality is that

no tool can do all three of these; at best you get two of the three. Typically, tools sacrifice precision for speed and number of checks.

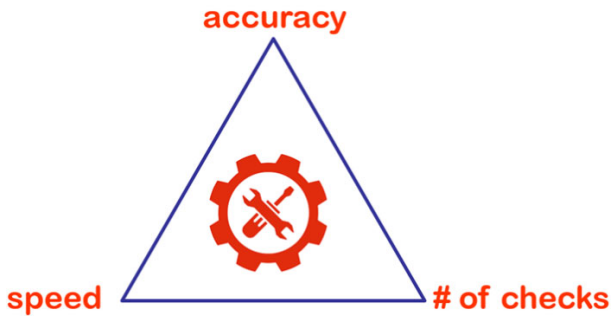

Figure 2: Three Dimensions of a Static Analysis Tools' Capabilities

Now, we will look at some of the design choices that a tool builder can make and they we will look at some examples of how these choices affect what the tool is capable of reporting.

Syntax vs. semantics: A tool can simply look at the syntax, the pattern of the program text – or semantics, the meaning of the program. Syntax-based tools are fast but can easily be misled. For example, they may simply look for a pattern in the text like the name of the often-dangerous Unix function `popen`. Semantic based tools on the other hand try to understand what the program is doing, so if they saw a call to `popen`, they would see if the parameter value could originate from the user. The additional information available in a semantics-based tool can reduce false alarms, but will run slower than a syntax-based tool.

Interprocedural analysis: If you are doing semantic analysis, you can evaluate each procedure (or function or method, whatever term you like) separately or can look at how they interact. There are cases where you can only tell if an operation in a function could be safe or not by understand what are the parameters that are passed to the function. You have to look at how the function is called to really know if there is a problem. This type of analysis can get complicated to implement and more expensive to run.

Whole program analysis: Another form of interprocedural analysis is called whole program analysis. Most tools will evaluate your program one source file at a time. This is not surprising as we usually compiler our code on file at a time. So, if a function is called from a function in another file, then you will not see the interprocedural effect. Few tools do whole program analysis as it is definitely more complex to build and slower to run.

Data flow analysis: Many weaknesses are only apparent if you understand how a variable's value is calculated. For example, is a subscript out of range? Or a will an integer calculation overflow? These questions can only be definitively determined if you can follow the flow of calculation in the program; this is called dataflow analysis. Dataflow analysis is complicated to implement and sometimes can be difficult, if not impossible to give an accurate result if the calculation is based on user input or if the program uses pointers and arrays. Most modern tools do some form of dataflow analysis, but limit its complexity to prevent it from slowing down the tool too much

Sound vs. approximate: We would like our tools to tell us the truth. This means that if there is no report for a line of code, then we would know that no weaknesses (of the type checked by this tool) are present on that line. And if there is a report for that line, we know that the report is true. Such a tool, in mathematical terms would be called *sound*. Unfortunately, no tool out there is sound; implementing such a tool is either too slow in the general case or just impossible (mathematically, we call that *undecidable*). Unfortunately, current static analysis the tools, especially the commercial ones, do not like to tell you when they are unsure of what they are doing.

The implication of these design choices for a tool is that they affect its speed and ability to accurately report what may be wrong with your code. Now we will look at a simple example and see how some of these issues come into play.

Consider a simple UNIX system call:

```
execl(cmd, NULL);
```

This is a call that overlays the currently running program in a process with a new program. In other words, it starts executing a new program that is stored in the file pointed to by the string `cmd`. The question is whether this line of code is safe or contains a weakness that might lead to a vulnerability?

A syntax-based tool would see this potentially dangerous call and flag it as dangerous. However, a semantic-based tool might evaluate the origin of the string pointed to by `cmd` and then make a more informed decision. Form example, if the code sequence was:

```
cmd = "/bin/ls";
execl(cmd, NULL);
```

Then we can that `cmd` refers to a constant string and did not originate from the user (i.e., from the attack surface) so this call to `execl` is safe. The `execl`

would simple execute the `ls` command. Now we will look at another variation:

```
fgets(*cmd, MAX, stdin);
execl(cmd, NULL);
```

Now that preceding line of code is one that reads the `cmd` string from user input. The user has complete control over what program is executed, i.e., there is a path from the attack surface to the use of `cmd` in the `execl`, so the tool should definitely say this code is dangerous. By allowing such a sequence, you are not controlling what the user is executing on your system.

We will make this example a little more interesting:

```
cmd=makecmd();
execl(cmd, NULL);
```

Now the value of cmd is a result of calling the `makecmd` function. Because we do not know what `makecmd` is doing, we cannot say whether the `execl` call would result in something dangerous. However, if the tool did interprocedural analysis, we might be able to know what is going on inside of `makecmd` and know, for example, that the value returned by `makecmd` was a constant string and did not originate from the attack surface. If function `makecmd` was in a different source code file than the function that contained the call to `execl`, then we would need whole program analysis to get sufficient information to understand whether the execl call was safe.

36.5 How do Static Analysis Tools Work?

When a tool analyzes your program, it may create a variety of different representations of the code. Some of these representations include ones that capture the relationships between classes, ones that model the program as a state machine, and, from the compiler world, ones that represent the control and data flow in the program. Figure 3 illustrates these three variations.

Tools that represent the data flow will try to understand the expressions that capture the value of variables using a technique called *symbolic evaluation.* Think of this technique as a kind of simulation of what values the variables might take. These abstract variables are expressions that describe a variable's type and possible values.

This kind of analysis is based on an extensive body of work in compiler theory and program analysis, using techniques like abstract interpretation and model checking. Graduate courses in this area are a great way to expand your knowledge of this technology.

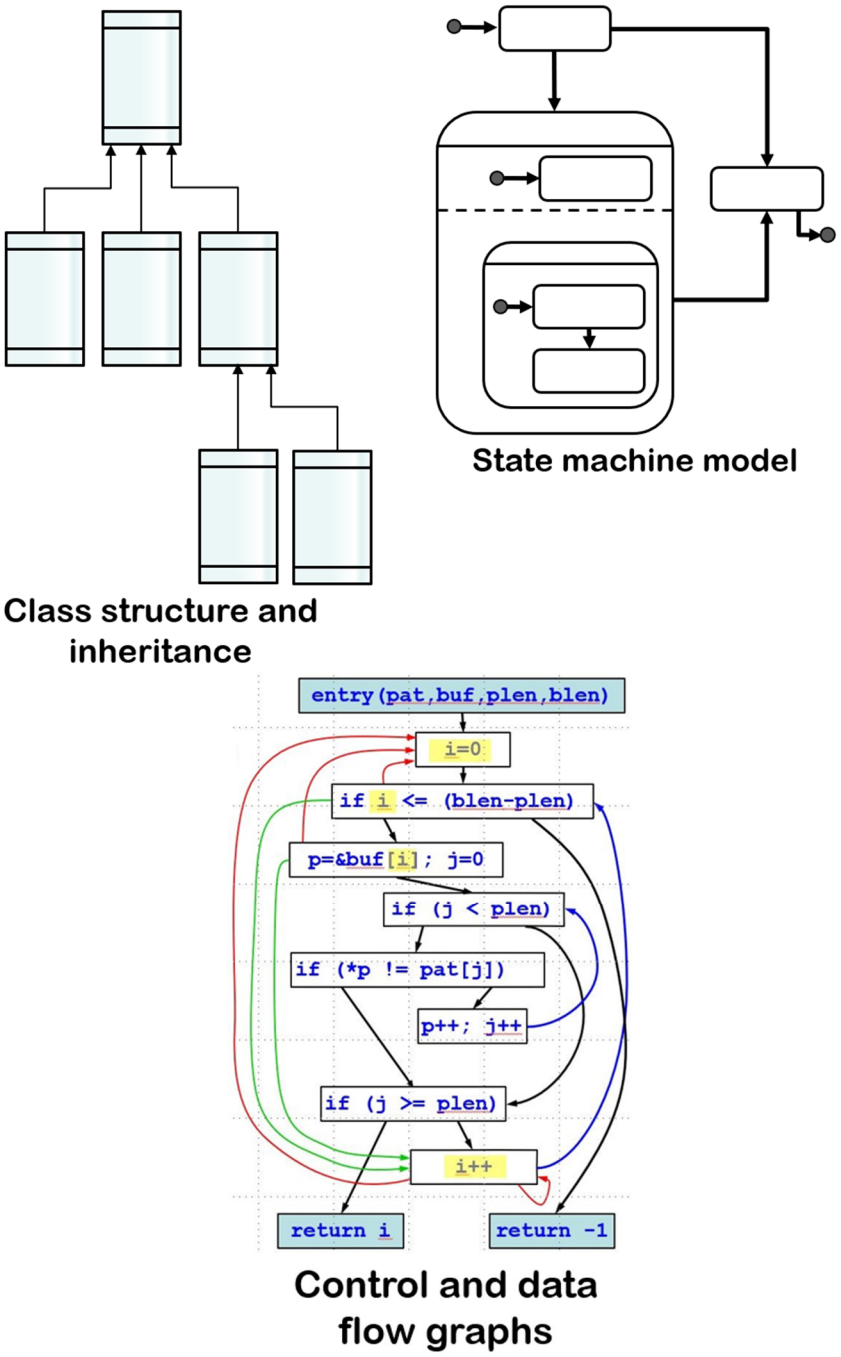

Figure 3: Three Program Code Analysis Models

One way that tools have to make results more accurate is to be *path sensitive.* This approach requires the analysis to keep state for each possible alternative for control flow. For example, does the tool separately track the possible states for the "then" and "else" branches of an if-statement? Or does it simply represent the union of the possible state changes from both branches? As with most code analysis topics, the increased accuracy comes at a cost of increased implementation complexity for the tool and decreased speed.

Another way that tools can make their results more accurate is to be *context sensitive.* This means tracking all the possible paths in the call graph so that you can do interprocedural analysis to see what effect the caller's state has on the callee. Again, this increased accuracy comes at a price.

36.6 Understanding What the Tools Tell You

When we run a static analysis tool, we get a list of weaknesses from the tool. And many tools will try to grade these weaknesses based on the severity, potential for exploit and how certain the tool is. The goal is to use this information to prioritize your efforts as to which weakness reports you investigate first (or ever).

You, as programmer or analyst, have the responsibility to look at these reports and decide if the report is correct, incorrect, or just not relevant or important enough to address. There is the challenge of dealing with a potentially long list of false positives and the unknown-unknowns, the lurking false negatives.

While these tools are automated, the decision of what to do with the results is controlled by you. As just mentioned, you have to first decide if the weakness is a true weakness, and then if it can lead to a vulnerability. Once you have made that determination, you have to decide if the risk associated with the vulnerability is serious enough that it is worth your time to fix. For the weaknesses that you will fix, you have to then identify the strategy for remediating them.

Something that experienced software assurance practitioners know is that there is no silver bullet, no single tool for a given programming language that will be best for all your needs.

Even though the commercial vendors want to lock you into their tool, and tell you that is all you need, experience analysts know that this is not the case. There are a variety of reasons this is true, including variation in computing environments, variation in algorithms or heuristics that are used by the tools, and the fact that new exploits are being developed all the time.

Now, we will look at a simple example of how this is true. Consider the Java code shown in Figure 4. When we ran this piece of Java code through several tools, we found a variety of non-overlapping reports from the different tools.

- PMD[3] told us that variable `y` was never used.
- Spotbugs[4] told us that you did not check this return value,
- Jlint[5] warned us about using "==" to compare String objects in Java. ("==" tells you if they are the same object, not if they have the same string value. For that you need to use the `equals` method.)
- Spotbugs also warned us that we did not close an I/O stream on exception.
- And ESC/Java[6] told us that the array index `i` is probably too big. The for-loop should test < length and not <=.

Note that we only list results from open course tools. We did not report any results here for commercial tools. That is because the tool vendors forbid anyone from publishing any results about how well they work. A terrible situation indeed![7]

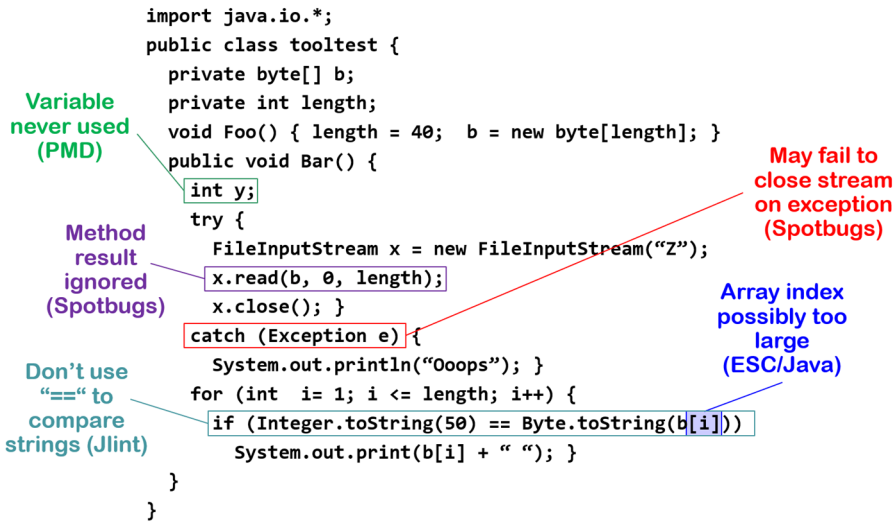

Figure 4: Weakness Identified by Different Static Analysis Tools

[3] https://pmd.github.io/
[4] https://spotbugs.github.io/
[5] https://jlint.sourceforge.net/
[6] https://en.wikipedia.org/wiki/ESC/Java (tool no longer supported)
[7] https://dwheeler.com/essays/dewitt-clause.html

Some of the things that analysis tools are good at finding include style problems, pointer errors, buffer overflows, races, resource leaks and some injections.

However, there are some things that might be difficult-but-not-impossible to find, such as use of weak passwords, incorrect access control list settings, or weak keys for cryptography. We note that there is recent research that is improving the checks for passwords and crypto issues, such as the CryptoGuard project from Virginia Tech[8].

And some things are just not appropriate for tools to check. These include a bad design, code that does not implement the design, configuration issues when you install the program, or errors based on the way different programs interact.

36.7 When to Use Static Analysis Tools

A basic question that we need to answer is when you should use static analysis tools? The answer is not that complicated: **always**. For **every project**. These tools are easy to use and even though they do not solve all your software security problems, they can make a noticeable improvement.

On a new project, you run the tool every time that you get ready to make a commit. By doing so, you knock down the weaknesses a few at a time at the moment that they are introduced into the code base.

We should discuss an important software project management issue for ongoing projects. What do you do with a legacy code base where the tools have not been used during development? If you applied an analysis tool to such a code base, it would be like suddenly turning on all the compiler warnings when they had never been used before. Most experienced programmers have been in this position at some point in their career.

If you turn on compiler warnings when they had never been used previously, or if start using a static analysis tools when they had not been previous used, you will find yourself faced with a *lot* of tool output. Really a lot. If you had 100,000 lines of code, you would get close to 100,000 reports! This is clearly an overwhelming amount of information, beyond what you can easily deal with.

However, not running static analysis tools because it is painful merely defers the pain and continues the bad practice of not using such tools. However, there is an effective approach to this problem, one that we have used successfully in our own software projects.

[8] `https://github.com/CryptoGuardOSS/cryptoguard`

You start with a simple but strict rule: every time someone commits a file, there must be a least *one fewer weakness* report from the analysis tool. The programmer's responsibility becomes fixing at least on weakness on every commit. If you follow this rule, you will get control over the number of weaknesses in a few months.

Why does this approach work in practice? It works because it is monotonic, so will steadily decrease. With each commit, there will always be fewer weaknesses with each commit. And with the shorter development and release cycles of today's software, you will make good progress.

But more important is how this interacts with the psychology of a typical programmer. For programmers, fixing these weaknesses is like eating potato chips: you cannot eat just one. A programmer will fix a weakness, then probably fix another and another, until they finally get bored or distracted after 5 or 10.

This approach is also called the *Boy Scout Rule,* "Leave the code cleaner than you found it".

The result is that you will clean up your code base more quickly than you thought, without making anyone too crazy.

```
int Find(char *pat, char *buf,
         unsigned int plen,
         unsigned int blen) {

   unsigned int i, j;
   char *p;

   i = 0;

   while (i <= (blen - plen)) {
      p = &buf[i];
      j = 0;
      while (j < plen) {
        if (*p != pat[j]) break;
        p++;
        j++;
      }
      if (j >= plen) return i;
      i++;
   }

   return -1;
}
```

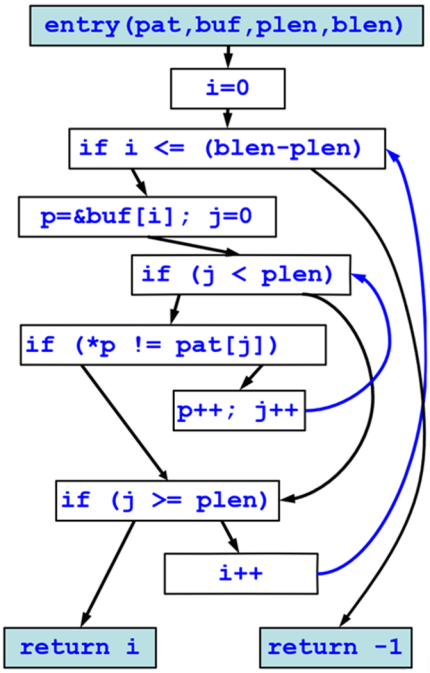

Figure 5: Simple Function Along with Control Flow Graph (CFG)

36.8 Basics of Code Analysis

A static analysis tools will use data structures to described the program that it is analyzing, better understand its structures, and to draw conclusions about its behavior. As we mentioned previously, basic data structures used by many code analyses include the control flow and dataflow graphs.

36.8.1 The Control Flow Graph

For control flow, we create the CFG, the *control flow graph* that captures execution order and branching. Figure 5 shows a simple function written in C along with its CFG.

The basic idea of the CFG is simple. We divide the programs into *basic blocks*, where each block represents a sequence of statements. Any place in the code that is the target of a jump or call is the start of a basic block. A jump ends a basic block. A basic block also ends at the start of the next block. This definition sounds a bit legalistic, but when you look back at the graph in Figure 5, it turns out to be pretty intuitive.

Take a moment to review the code and CFG in Figure 5. See if you can map each node in the CFG to its corresponding block of C.

The forward control flow edges are shown in black and the backwards control flow edges are shown in blue.

36.8.2 The Dataflow Graph

For data flow, we annotate the CFG with dataflow information to reflect how data flows through the computation. For example, looking at Figure 5, we see the variable `i` used in the program as an array subscript. The dataflow graph (DFG) will allow the tool to describe how the variable is used and modified, perhaps allowing the tool to decide if the subscript can ever reference beyond the bounds of the array.

The DFG starts with the CFG and then annotates with graph with additional edges to show how data flows through the program. The dataflow edges flow from the place where a variable is used to a place where it is modified.

In Figure *6*, we highlight in yellow the places that variable `i` is used or modified in the code. We have also added dataflow edges for variable `i`, with the red edges showing how data values flow forward through the program's execution and green edges showing flows backwards due to loops. These backwards edges describe loop carried dependencies, which describe a place where a value is set during one iteration of a loop and then used during a subsequent one. For example, the `i++` statement at the bottom of the loop

modifies `i`, which is then used at the top of the loop if the first `if` statement during the next iteration of the loop.

You could add similar edges for variables `j`, another integer used as an array subscript, and `p`, a pointer used to also reference the contents of one of the character arrays.

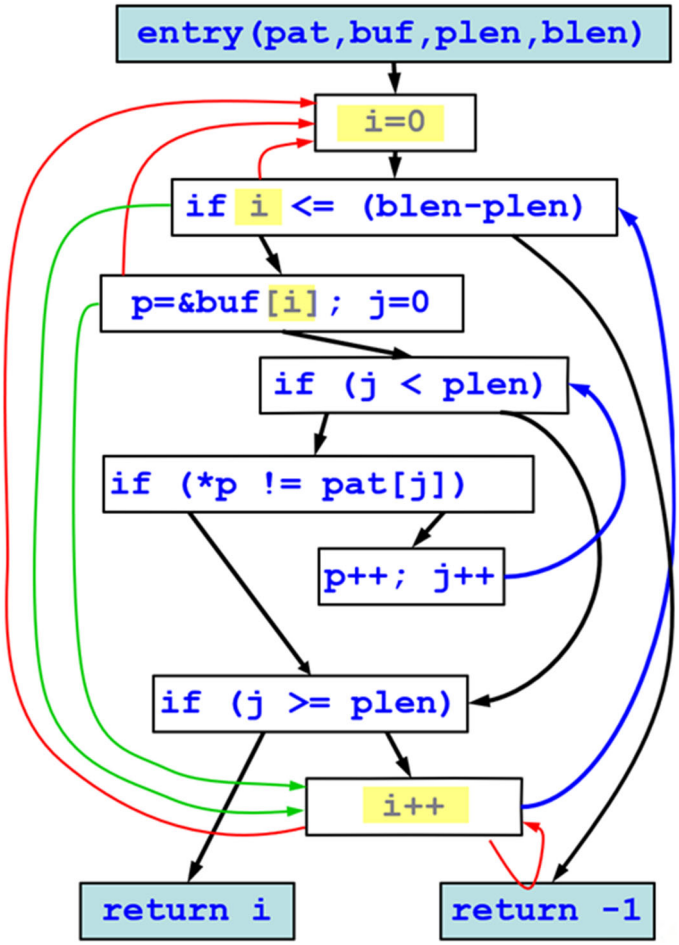

Figure 6: Dataflow Graph with Edges for Variable "`i`"

This sample function exhibits an interesting property that makes code analysis more complex. Note that the contents of the array `buf` are referenced by both subscripting, as done with `buf[i]`, and pointer dereferencing, as done with `*p`. When the function executes the statement:

```
p=&buf[i];
```

it is creating two different ways to reference the contents of the array. The pointer `p` is set to the address of the `i`th element of `buf`. This creation of two

different way so referencing the same memory is called *aliasing*. It happens commonly in programs and makes the job of the analysis tools more difficult.

A static analysis tool might use the dataflow information in the program to infer the values that variables, such as for pointers and integer array indices, to try to determine their potential range of values.

```
int Find(char *pat, char *buf,
         unsigned int plen,
         unsigned int blen) {

   unsigned int i, j;
   char *p;

   i = 0;                              i:[0,0]

   while (i <= (blen - plen)) {        i:[0,blen-plen+1]
      p = &buf[i];                     i:[0,blen-plen]
                                       p:buf[0,blen-plen]
      j = 0;                           j:[0,0]
      while (j < plen) {               j:[0,plen]
        if (*p != pat[j]) break;       j:[0,plen-1]
                                       p:[buf[0,blen-plen+plen-1]
        p++;                           p:[buf[1,blen-plen+plen]
        j++;                           j:[1,plen]
      }
      if (j >= plen) return i;         j:[0,plen]
                                       i:[0,blen-plen]
      i++;                             i:[1,blen-plen+1]
   }
   return -1;
}
```

Figure 7: Sample Function with Ranges Shown for Variables `i`, `j`, and `p`

In Figure 7, we see our sample function with the ranges ([lower bound, upper bound]) shown for variables `i`, `j`, and `p` for each line of code. The values represent the variables after the corresponding statement has executed. For example, after `i=0` has executed, the variable `i` will have a value of exactly and only zero.

The value for `p` after the `if (*p != pat[j])` has executed is a bit complex. Take a careful look at the code and previous values computed to determine why `p` can have an upper bound of `blen-plen+plen-1`. Note that this expression simplifies to `blen-1`.

While we were able to provide range expressions for all the lines of code in this sample functions, this is not always easy to do for real code. There are a variety of code features that can make such analyses slow, imprecise, or even infeasible.

Recursion and deep loop nesting can make such expressions more expensive (so slower) to compute, resulting in potentially less accuracy.

Complex subscript and pointer expressions can make such determinations slow to compute or even impossible. For example, if the loop controlling expression has non-affine[9] expression, the complexity of calculating the ranges can grow significantly.

Pointers to functions in the program, especially if calculating the values of those pointers is complex, can make such calculations more difficult. Function pointers appear explicitly in C and C++, but can also appear implicitly with the use of virtual functions.

And programs that have concurrency (multi-threading) make almost all aspects of static analysis extremely difficult.

36.9 Summary

- Learned a little history of static analysis tools
- Understood how static analysis tools work
- Understood the limitations of these tools
- Understood the role of the analyst (you!) when using these tools.
- Learned about control and data flow graphs used by some tool

36.10 Exercises

1. What is the significant difference between the `-Wall` warning option for the gcc/clang C compiler and the corresponding `/Wall` option for Microsoft Visual Studio C compiler?
2. In this code fragment
   ```
   fgets(*cmd, MAX, stdin);
   execl(cmd, NULL);
   ```
   the variable cmd might originate from the attack surface (user input). Given an example of how an attacker might use this fact to exploit this vulnerability.
3. How would you integrate the use of static analysis tools into a continuous integration[10] process?
4. Figure 7 shows a sample function along with the ranges of three variables that the static analysis tool might have computed. The most complex range expression for a line of code in Figure 7 is `blen-`

[9] Affine expressions are simple linear expressions of the form *ax* +*b*, when *a* and **b** are constants.

[10] `https://en.wikipedia.org/wiki/Continuous_integration`

`plen+plen-1`. Describe how that value is computed from previously computed values.

5. The function in Figure 7 has a subtle but serious bug in it. Assume that the function is called correctly. That is to say, the pointers `buf` and `pat` correctly point to value NULL-terminated strings for the buffer and search pattern, and that these buffers are of sufficient size to hold these strings. And the values `blen` and `plen` correctly describe the length of each of these two strings. Given these assumptions, there is still a bug. Find this bug.