

Program Analysis

Thea Rossman
Jan 6, 2022

Logistics

- Please make sure you're on Slack (Canvas -> Slack tab)
- Please fill out [intro survey](#) if you haven't yet
- Week 1 exercises coming out today; due next Monday. (Write your answers directly into Gradescope!)
- Follow-up on course content:
 - Range of CS backgrounds in this class
 - Other options: material Ryan has posted on his [website](#); Rust [website](#); CS242 (programming languages); CS295 (program analysis); etc.

Content

- Today (+ exercises): what are some tools that we can use to find mistakes in C/C++ code? What are their limitations?
- Next week: How do other languages address some shortcomings of C?

How can we find bugs in a program?

Dynamic Analysis

Dynamic analysis: high-level

- Run the program, watch what it does, and look for problematic behavior
- Can find problems, but only if the program exhibits problematic behavior on the inputs you use to test. (Separately, some tools only check for certain types of issues.)
- Commonly combined with techniques to run the program with lots of different test inputs (e.g. fuzzing), yet this still can't give us any assurances that code is bug-free
- Dynamic analysis is great! Test your code! *and* understand the limitations!

Dynamic analysis tool: Valgrind

- Instruments *binaries* on the fly

```
int main() {  
    char *buf = (char*)malloc(8);  
    buf[16] = 'a';  
}
```

(compiler)

```
mov edi, 8  
call malloc  
mov QWORD PTR [rbp-8], rax
```

```
mov rax, QWORD PTR [rbp-8]  
add rax, 16  
mov BYTE PTR [rax], 97
```

(valgrind)

```
mov edi, 8  
call valgrind_malloc  
mov QWORD PTR [rbp-8], rax  
record memory write ^
```

```
mov rax, QWORD PTR [rbp-8]  
record memory read ^  
add rax, 16  
mov BYTE PTR [rax], 97  
record memory write ^
```

Invalid write of size 4
(writing to the heap, but it's not
inside any heap allocation that was
previously made)

Valgrind (summary)

- Takes in compiled binary (executable)
- Disassembles binary into intermediate, assembly/assembly-like representation
- Turns this ^ back into machine code (re-compiler); one small block at a time during execution
- ...while “instrumenting”: modifying instructions or inserting “analysis” code
 - *For example, Valgrind’s [memcheck](#) stores some “shadow” metadata about heap memory accessed by your code — for example, bits indicating “this has been freed” or “this has been initialized”. Using these, it can detect issues like double-freed heap blocks or uninitialized values.*

Valgrind

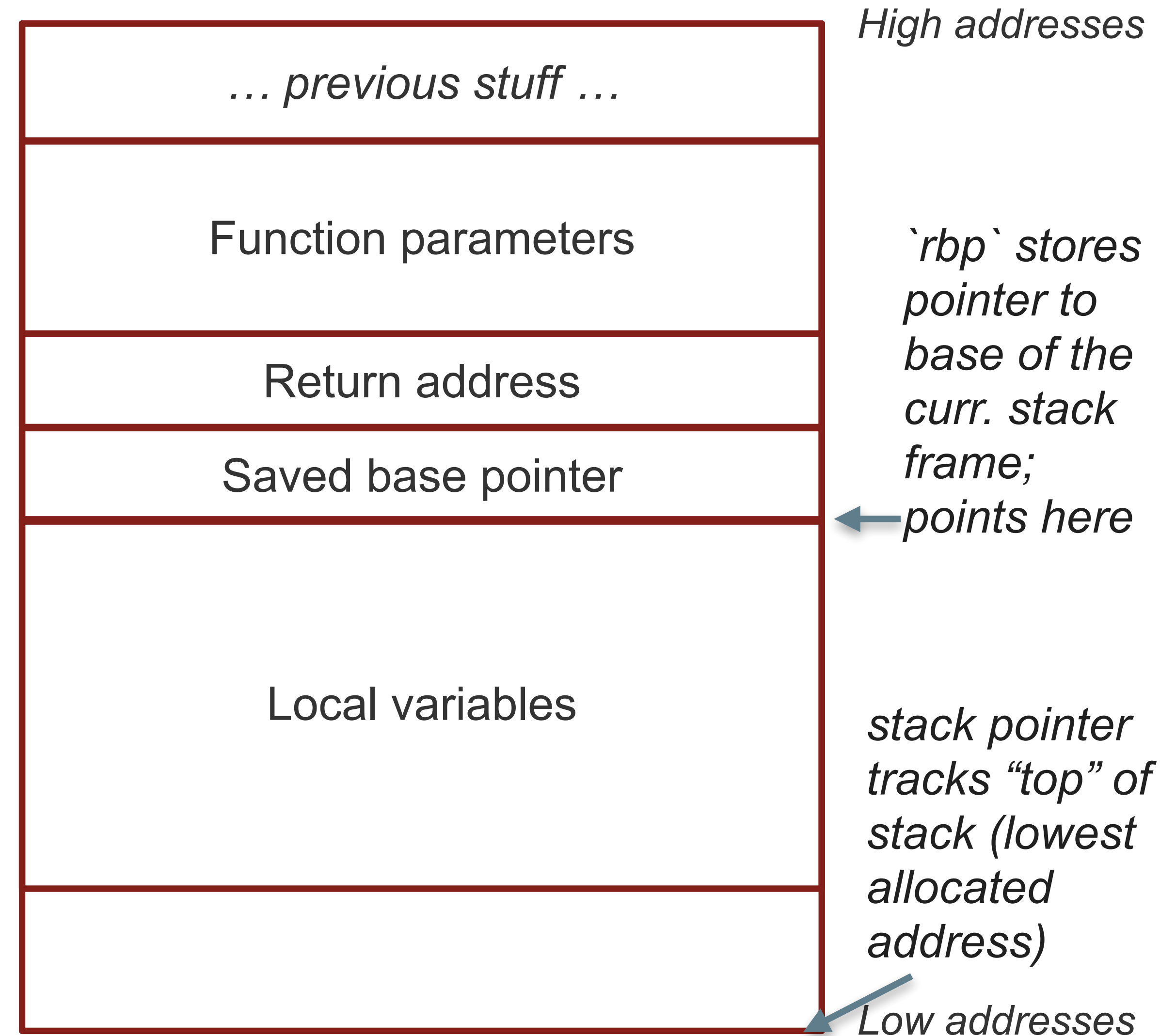
- Works with any binary compiled by any compiler (even if you don't have source code available!)
- Downside: not a lot of information is available in binaries...

From last lecture: anatomy of a stack frame

- Stack grows DOWN (from higher to lower addresses) as functions are *called* (new stack frames created) or require local variables.
- Stack is just a chunk of memory — no information in binary about how it's split into variables.
- Remember: Writing to local stack buffers goes UP (from lower to higher addresses)

```
mov edi, 8  
call malloc  
mov QWORD PTR [rbp-8], rax  
...
```

“write to `buf`” becomes “write to the memory location `8` bytes below the base of the stack”



Valgrind

- Works with any binary compiled by any compiler (even if you don't have source code available!)
- Downside: not a lot of information is available in binaries
 - E.g. the stack is just a chunk of memory. You might be able to observe that the stack pointer grows up/down, but no information about how it's divided into variables.
 - => cannot detect stack-based buffer overflows!

LLVM Sanitizers

- Same idea, but instrument *source code*
- Implemented as part of the LLVM compiler suite (e.g. clang)
- Because more information is available pre-compilation, there is a lot more analysis that sanitizers can do (and they're also easier to implement)

```
int main() {  
    char buf[8];  
    Record stack buffer "buf" with size 8  
    buf[16] = 'a';  
    Record write to "buf" with offset 16  
}
```

LLVM Sanitizers

- AddressSanitizer
 - Finds use of improper memory addresses: out of bounds memory accesses, double free, use after free
- LeakSanitizer
 - Finds memory leaks
- MemorySanitizer
 - Finds use of uninitialized memory
- UndefinedBehaviorSanitizer
 - Finds usage of null pointers, integer/float overflow, etc
- ThreadSanitizer
 - Finds improper usage of threads (second half of CS 110)
- More...

Cool! Let's sanitize *all* the code!! 🏍️🔥 100



Fundamental limitation of dynamic analysis

- Dynamic analysis can only report bad behavior that *actually happened*
- If your program worked fine with the input you provided, but it might do bad things in certain edge cases, dynamic analysis cannot tell you anything about that

```
#include <stdio.h>
#include <string.h>
int main() {
    char s[100];
    int i;
    printf("\nEnter a string : ");
    gets(s);
    for (i = 0; s[i]!='\0'; i++) {
        if(s[i] >= 'a' && s[i] <= 'z') {
            s[i] = s[i] -32;
        }
    }
    printf("\nString in Upper Case = %s", s);
    return 0;
}
```

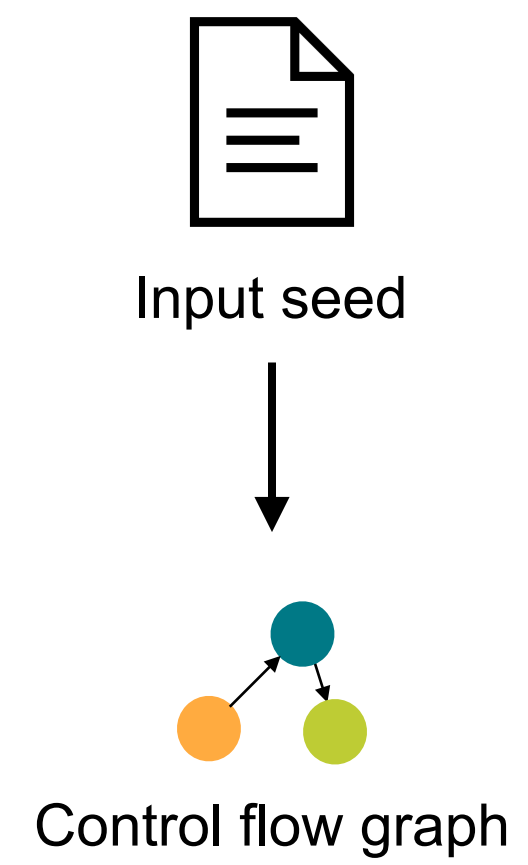
How can we find weird edge cases?

Fuzzing

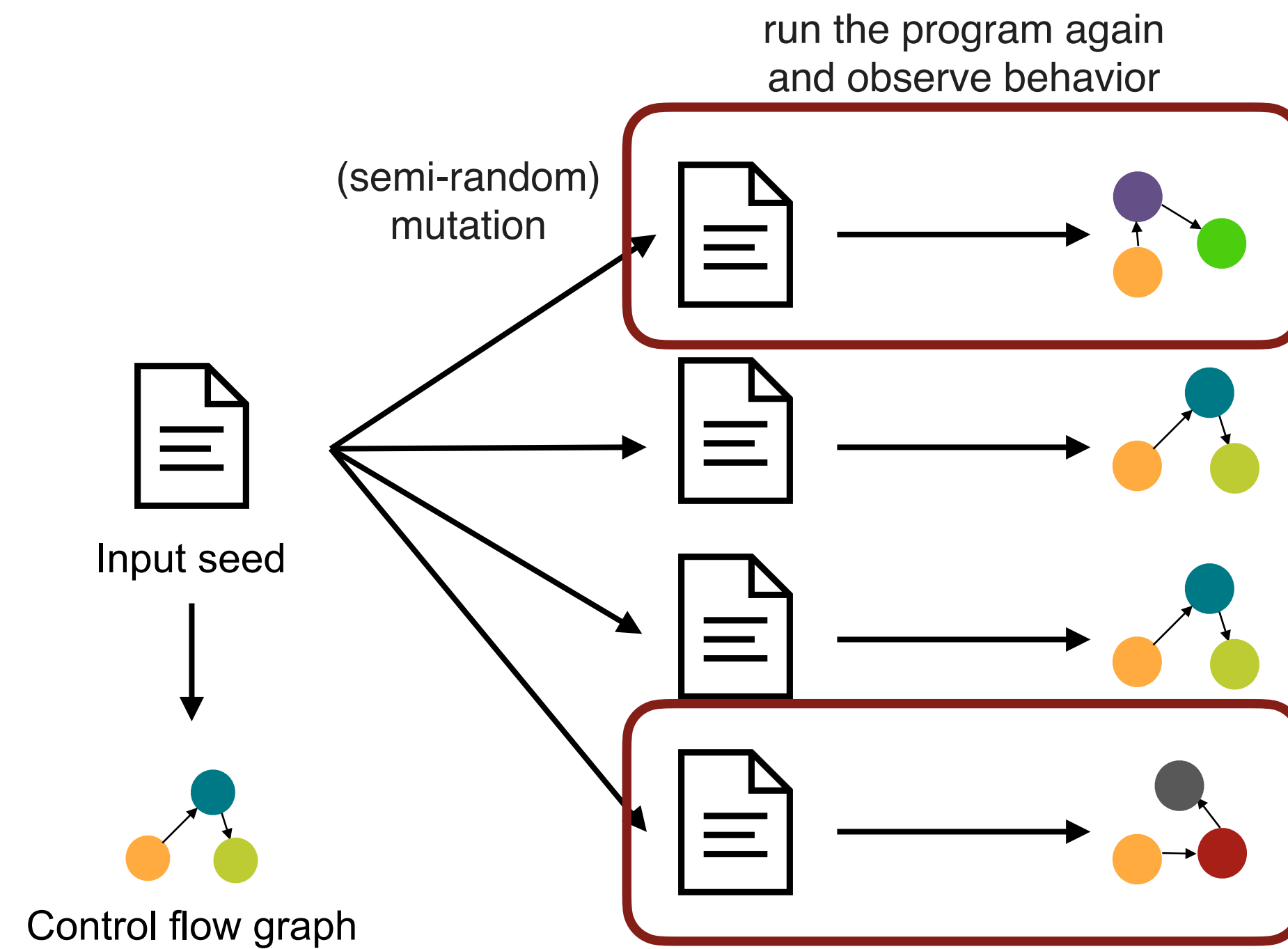


Input seed

Fuzzing

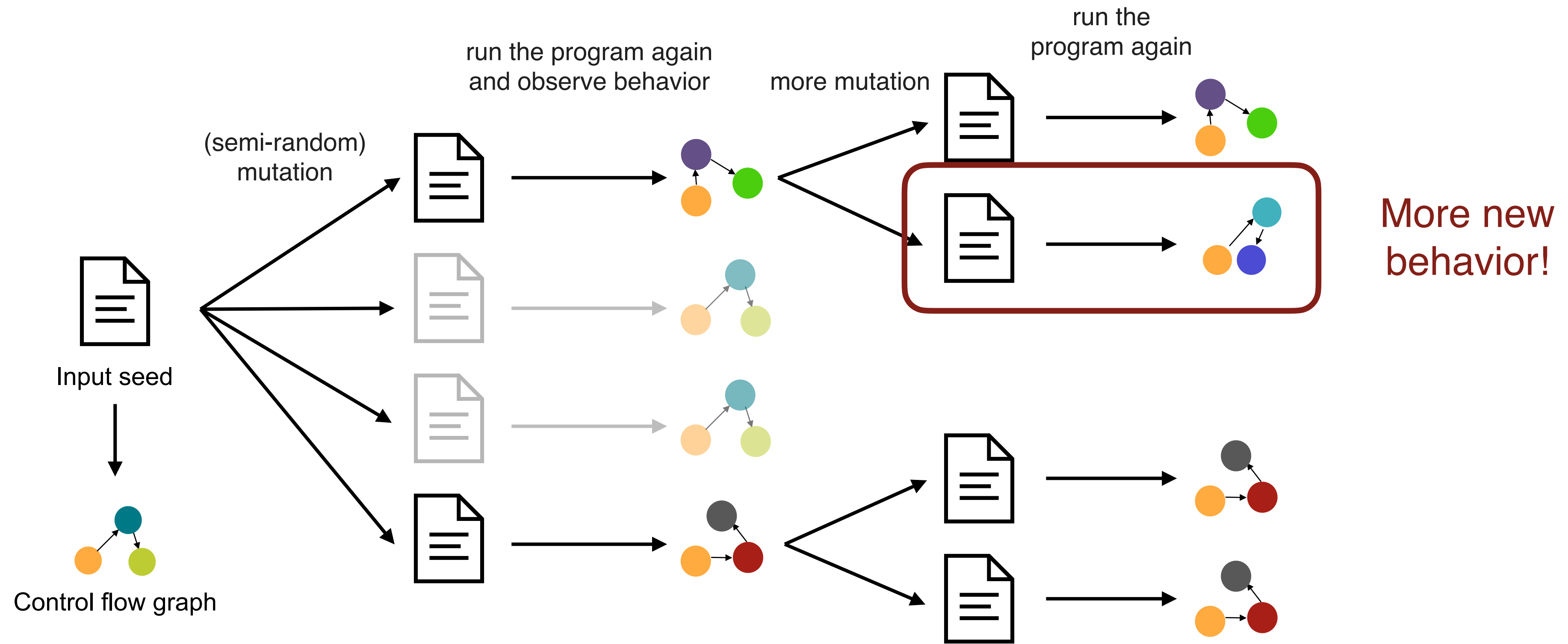


Fuzzing



These inputs
made the
program do
new things!

Fuzzing



Continue this process forever...

Fuzzing

- Very simple but *extremely* effective
- Most common fuzzers: [AFL](#) and [libfuzzer](#)
- Still, cannot provide any guarantees that a program is bug-free (if the fuzzer didn't find anything in 24 hours, maybe we just didn't run it long enough)
- Google [OSS-Fuzz](#) is a large cluster that fuzzes open-source software 24/7

Static Analysis

You Be the Static Analyzer: Round 1

You want to write a tool to help people writing code like this. What do you do?

```
#include <stdio.h>
#include <string.h>
int main() {
    char s[100];
    int i;
    printf("\nEnter a string : ");
    gets(s);
    for (i = 0; s[i]!='\0'; i++) {
        if(s[i] >= 'a' && s[i] <= 'z') {
            s[i] = s[i] -32;
        }
    }
    printf("\nString in Upper Case = %s", s);
    return 0;
}
```

Basic static analysis (“linting”)

Stephen C. Johnson, a computer scientist at Bell Labs, came up with lint in 1978... The term "lint" was derived from the name of the tiny bits of fiber and fluff shed by clothing, as the command should act like a dryer machine lint trap, detecting small errors with big effects.

[https://en.wikipedia.org/wiki/Lint_\(software\)](https://en.wikipedia.org/wiki/Lint_(software))

- Linters employ very simple techniques (e.g. ctrl+f) to find obvious mistakes
- The person running the linter can configure a set of rules to enforce
 - Rules are intended to improve the style of the codebase
 - Just because there is a linter error doesn't mean the code is broken (e.g. it's possible to call `strcpy()` without introducing bugs, but many linters will complain if you call it)
- Common C/C++ linter: [clang-tidy](#)
 - Can even auto-fix many of the issues!

You Be the Static Analyzer: Round 2

You want to write a tool to help people writing code like this. What do you do?


```
void printToUpper(const char *str) {
    char *upper = strdup(str);
    for (int i = 0; str[i] != '\0'; i++) {
        if(str[i] >= 'a' && str[i] <= 'z') {
            upper[i] = str[i] - ('a' - 'A');
        }
    }
    printf("%s\n", upper);
    free(upper);
}
```

```
int main(int argc, char *argv[]) {
    printf("Enter a string to make uppercase,
or type \"quit\" to quit:\n");
    char input[512];
    // safely read input string
    fgets(input, sizeof(input), stdin);
    char *toMakeUppercase;
    if (strcmp(input, "quit") != 0) {
        toMakeUppercase = input;
    }
    printToUpper(toMakeUppercase);
}
```

Dataflow analysis

We can trace through how the program might execute, keeping track of possible variable values

```
int main(int argc, char *argv[]) {  
    printf("Enter a string to make uppercase,  
or type \"quit\" to quit:\n");  
    char input[512];  
    // safely read input string  
    fgets(input, sizeof(input), stdin);  
    char *toMakeUppercase; _____ toMakeUppercase = {uninitialized}  
    if (strcmp(input, "quit") != 0) {  
        toMakeUppercase = input;  
    }  
    printToUpper(toMakeUppercase);  
}
```



clang-tidy will do dataflow analysis too!

Dataflow analysis

We can trace through how the program might execute, keeping track of possible variable values

```
int main(int argc, char *argv[]) {  
    printf("Enter a string to make uppercase,  
or type \"quit\" to quit:\n");  
    char input[512];  
    // safely read input string  
    fgets(input, sizeof(input), stdin);  
    char *toMakeUppercase;  
    if (strcmp(input, "quit") != 0) {  
        toMakeUppercase = input;  
    }  
    printToUpper(toMakeUppercase);  
}
```

toMakeUppercase = {uninitialized}

Dataflow analysis

We can trace through how the program might execute, keeping track of possible variable values

```
int main(int argc, char *argv[]) {  
    printf("Enter a string to make uppercase,  
or type \"quit\" to quit:\n");  
    char input[512];  
    // safely read input string  
    fgets(input, sizeof(input), stdin);  
    char *toMakeUppercase;  
    if (strcmp(input, "quit") != 0) {  
        toMakeUppercase = input;  
    }  
    printToUpper(toMakeUppercase);  
}
```

toMakeUppercase = {*uninitialized*, input}

printToUpper called with a possibly uninitialized argument!

Dataflow analysis: very powerful!

You want to write a tool to help people writing code like this. What do you do?

```
int main(int argc, char *argv[]) {  
    // Goal: parse out a string between brackets  
    // (e.g. "[target string]" -> "target string")
```

```
    char *parsed = strdup(argv[1]);
```

```
    // Find open bracket
```

```
    char *open_bracket = strchr(parsed, '[');
```

```
    if (open_bracket == NULL) {  
        printf("Malformed input!\n");  
        return 1;  
    }
```

**Common mistake: early
return fails to clean up
resources**

```
    // Make the output string start after the open bracket  
    parsed = open_bracket + 1;
```

```
    // Find the close bracket
```

```
    char *close_bracket = strchr(parsed, ']');
```

```
    if (close_bracket == NULL) {  
        printf("Malformed input!\n");  
        return 1;  
    }
```

```
    // Replace the close bracket with a null
```

```
    // terminator to end the parsed string there
```

```
    *close_bracket = '\0';
```

```
    printf("Parsed string: %s\n", parsed);
```

```
    free(parsed);
```

```
    return 0;
```

```
}
```

Dataflow analysis: very powerful!

Liveness analysis: observe when variables go away, and make sure they're cleaned up appropriately

```
int main(int argc, char *argv[]) {
    // Goal: parse out a string between brackets
    // (e.g. "[target string]" -> "target string")

    char *parsed = strdup(argv[1]);
    // _____ parsed = {heap allocation}

    // Find open bracket
    char *open_bracket = strchr(parsed, '[');
    if (open_bracket == NULL) {
        printf("Malformed input!\n");
        return 1;
    }

    // Make the output string start after the open bracket
    parsed = open_bracket + 1;
```

```
    // Find the close bracket
    char *close_bracket = strchr(parsed, ']');
    if (close_bracket == NULL) {
        printf("Malformed input!\n");
        return 1;
    }

    // Replace the close bracket with a null
    // terminator to end the parsed string there
    *close_bracket = '\0';

    printf("Parsed string: %s\n", parsed);
    free(parsed);
    return 0;
}
```


Dataflow analysis: very powerful!

Liveness analysis: observe when variables go away, and make sure they're cleaned up appropriately

```
int main(int argc, char *argv[]) {
    // Goal: parse out a string between brackets
    // (e.g. "[target string]" -> "target string")

    char *parsed = strdup(argv[1]);

    // Find open bracket
    char *open_bracket = strchr(parsed, '[');
    if (open_bracket == NULL) {
        printf("Malformed input!\n");
        return 1;
    }
    // Make the output string start after the open bracket
    parsed = open_bracket + 1;
```

parsed = {heap allocation}
**parsed is no longer live, but is still
a heap allocation!**

```
    // Find the close bracket
    char *close_bracket = strchr(parsed, ']');
    if (close_bracket == NULL) {
        printf("Malformed input!\n");
        return 1;
    }

    // Replace the close bracket with a null
    // terminator to end the parsed string there
    *close_bracket = '\0';

    printf("Parsed string: %s\n", parsed);
    free(parsed);
    return 0;
}
```

Dataflow analysis: works across functions

Tracking calls to functions is no different from tracing paths through if statements

```
void freeSometimes(void *buf) {  
    if (rand() == 1) {  
        return;  
    }  
    free(buf);  
}  
  
int main() {  
    void *buf = malloc(8);  
    freeSometimes(buf); buf = {heap allocation}  
    return 0;  
}
```


Dataflow analysis: works across functions

Tracking calls to functions is no different from tracing paths through if statements

```
void freeSometimes(void *buf) {  
    if (rand() == 1) { buf = {heap allocation}  
        return;  
    }  
    free(buf);  
}  
  
int main() {  
    void *buf = malloc(8);  
    freeSometimes(buf);  
    return 0;  
}
```

Dataflow analysis: works across functions

Tracking calls to functions is no different from tracing paths through if statements

```
void freeSometimes(void *buf) {  
    if (rand() == 1) {  
        return;                      buf = {heap allocation}  
    }  
    free(buf);                      buf = {heap allocation}  
}  
  
int main() {  
    void *buf = malloc(8);  
    freeSometimes(buf);  
    return 0;  
}
```

Dataflow analysis: works across functions

Tracking calls to functions is no different from tracing paths through if statements

```
void freeSometimes(void *buf) {  
    if (rand() == 1) {  
        return;                      buf = {heap allocation}  
    }  
    free(buf);                      buf = {freed allocation}  
}  
  
int main() {  
    void *buf = malloc(8);  
    freeSometimes(buf);  
    return 0;  
}
```

Dataflow analysis: works across functions

Tracking calls to functions is no different from tracing paths through if statements

```
void freeSometimes(void *buf) {  
    if (rand() == 1) {  
        return;  
    }  
    free(buf);  
}  
  
int main() {  
    void *buf = malloc(8);  
    freeSometimes(buf);  
    return 0;                      buf = {heap allocation, freed allocation}  
}
```

Dataflow analysis: works across functions

Tracking calls to functions is no different from tracing paths through if statements

```
broken.c:13:5: warning: Potential leak of memory pointed to by 'buf' [clang-analyzer-  
unix.Malloc]
```

```
    return 0;
```

```
    ^
```

```
broken.c:11:17: note: Memory is allocated
```

```
    void *buf = malloc(8);
```

```
            ^
```

```
broken.c:13:5: note: Potential leak of memory pointed to by 'buf'
```

```
    return 0;
```

```
    ^
```

Limitations

- False positives
 - Dataflow analysis will follow each branch, even if it's impossible for some condition to be true in real life
 - False positives are the Achilles' heel of static analysis. Need a good signal/noise ratio or else no one will use your analyzer
- Need to limit scope to get reasonable performance
 - Many static analyzers only analyze a single file at a time: they don't do dataflow analysis into/out of functions elsewhere in the codebase
 - If you have a huge codebase, loops, tons of conditions, etc., dataflow analysis can get unwieldy.

Take CS 243 for more info!

Cool! Let's tidy *all* the code!! 🏎️🔥 100



Low-hanging fruit #1

```
int main(int argc, char *argv[]) {  
    char *message = strchr(argv[1], 'a');  
    printf("%s\n", message);  
}
```

Low-hanging fruit #1

🍓 clang-tidy easy.c

🍓 cppcheck easy.c *no output here means no issues found*

Checking easy.c ...

🍓 scan-build clang-11 -Wall easy.c

scan-build: Using '/usr/local/Cellar/llvm/11.0.0_1/bin/clang-11' for static analysis

scan-build: Analysis run complete.

scan-build: Removing directory '/var/folders/6_/jdc6ljyd5n795x1xl8drptm80000gn/T/scan-build-2021-04-01-002241-43549-1' because it contains no reports.

scan-build: No bugs found.

How do we fix this?

- Okay, I'll just make sure programs can handle receiving NULL from strchr
- But what if the program is calling strchr on a string that is guaranteed to have the character they're looking for? (i.e. strchr will for sure not return NULL)
- And what about all the other functions that can potentially return NULL for one reason or another?
- And what about...

Low-hanging fruit #2

```
int main(int argc, char *argv[]) {  
    char buf[16];  
    strncpy(buf, argv[1], sizeof(buf));  
    printf("%s\n", buf);  
}
```

The **strncpy()** function is similar, except that at most n bytes of *src* are copied. **Warning:** If there is no null byte among the first n bytes of *src*, the string placed in *dest* will not be null-terminated.

<https://linux.die.net/man/3/strncpy>

Similar real-world example: <https://en.wikipedia.org/wiki/Heartbleed>

Low-hanging fruit #2

🍓 clang-tidy easy.c

🍓 cppcheck easy.c

Checking easy.c ...

🍓 scan-build clang-11 -Wall easy.c

scan-build: Using '/usr/local/Cellar/llvm/11.0.0_1/bin/clang-11' for static analysis

scan-build: Analysis run complete.

scan-build: Removing directory '/var/folders/6_/jdc6ljyd5n795x1xl8drptm80000gn/T/scan-build-2021-04-01-002241-43549-1' because it contains no reports.

scan-build: No bugs found.

How do we fix this?

- Okay, I'll just make sure programs add a null terminator after calling `strncpy`
- But what if the program actually uses the copied "string" as a character array instead of a null-terminated string (i.e. the code is actually fine)?
- And how are you going to track down every function that depends on the string having a null terminator?
- Note: outright banning `strncpy()` might be a better idea, but there are still other ways we could end up with a `char*` that is not a null-terminated string

Fundamental limitations of static analysis

- If you can only look at a few lines of code, it's hard to tell (without broader context) whether that code is safe
- Getting broader context is impossible in the general case (see: “the halting problem”)
 - We can guesstimate what values get passed around in a program using dataflow analysis, and we can guesstimate how they get used, but it breaks down when code gets complicated
- You can always add more specific things to check for, but there will always be other ways to mess up
- Begin to think about: is there some way we can make it easier to verify small snippets of code in isolation, without broader context?
 - *More next week: This general idea is a key motivation for Rust!*

Takeaways

- If you are writing C/C++, you should absolutely be running sanitizers, fuzzers, and static analyzers
 - You should understand the limitations of these tools, but...
 - Just because they are limited does not mean they aren't helpful
- If you are in a position to use languages with more robust protections, you should!

For next week

- Take 10 minutes to look through this buggy vector implementation: <https://web.stanford.edu/class/cs110/lecture-notes/lecture-03/>
 - Try to find as many bugs as you can
- Week 1 exercises due Monday (released today)