

# Program Analysis

Ryan Eberhardt and Julio Ballista  
April 1, 2021

# Logistics

- Please make sure you're on Slack
- Fill out the intro survey if you haven't already
- Week 1 exercises + survey coming out today
- Today: 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 the shortcomings of C?

How can we find bugs in a program?

# How can we find bugs in a program?

- Dynamic analysis: run the program and watch what it does
- Static analysis: read the source code

# Dynamic Analysis

# 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

- 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 hunk of memory. No information about how it's allocated into variables
  - => cannot detect stack-based buffer overflows!

# LLVM Sanitizers

- Same idea, but instrument *source code* (kind of)
- 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;
}
```

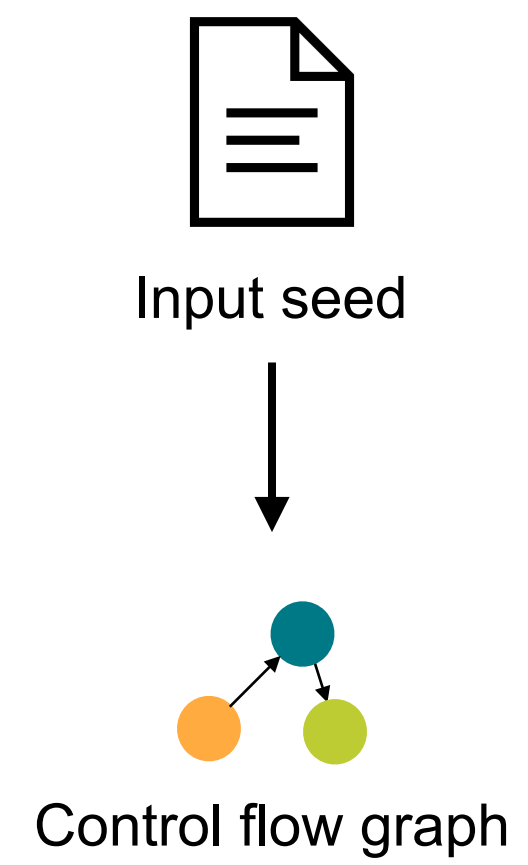
How can we find weird edge cases?

# Fuzzing

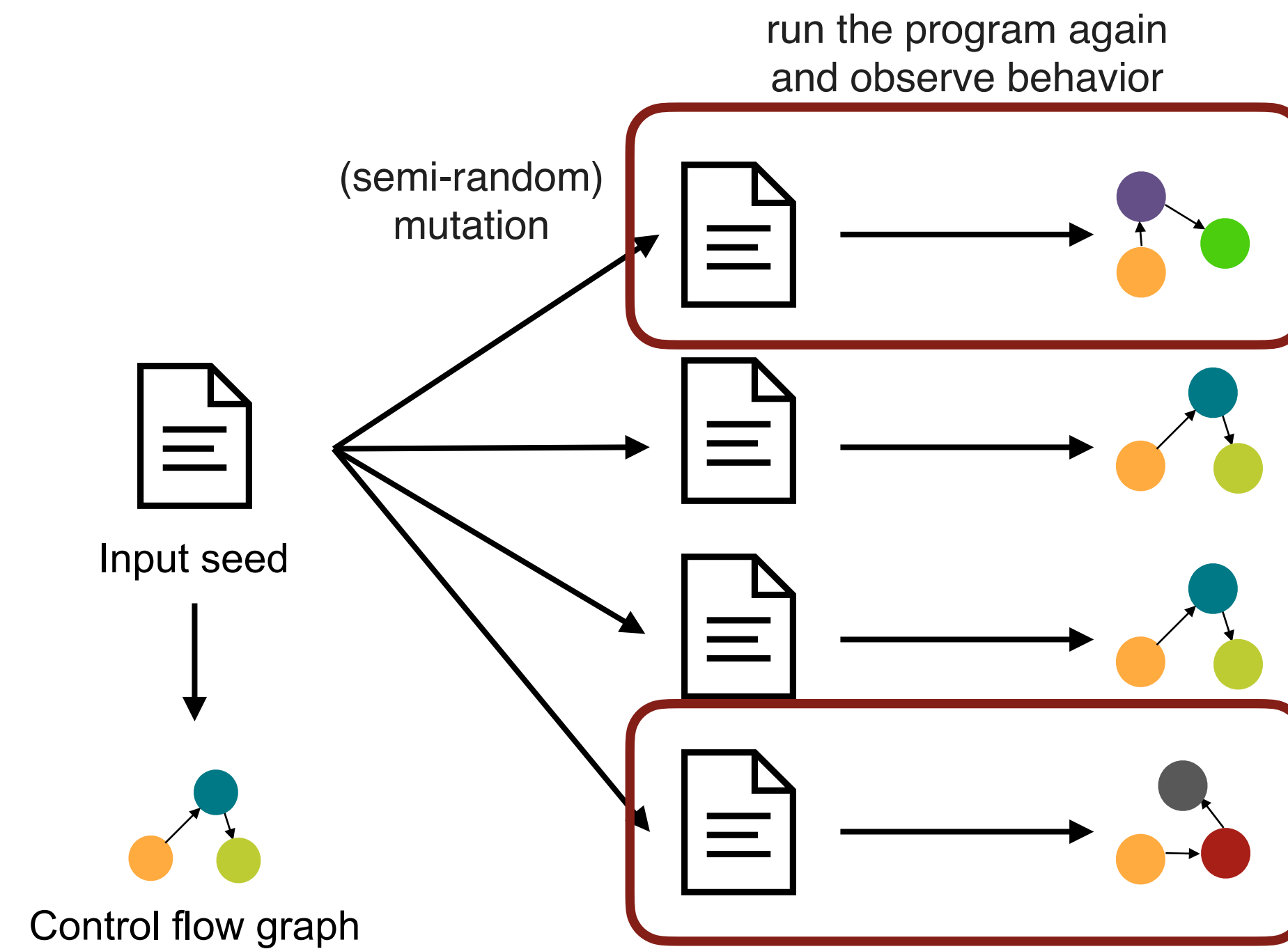


Input seed

# Fuzzing

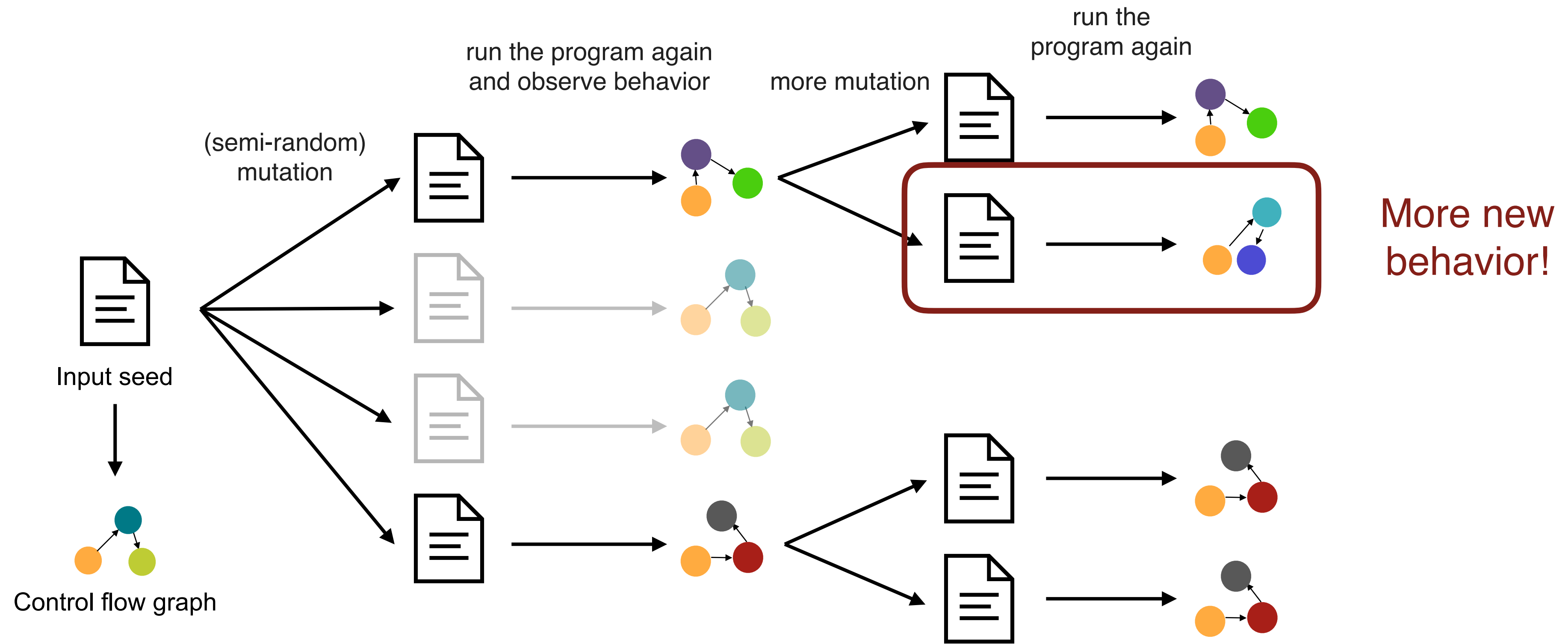


# Fuzzing



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);  
}
```



# 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
- 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

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

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);  
}
```



# 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\_/jdc6ljyd5n795x1x18drptm80000gn/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 `strchr()` would 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

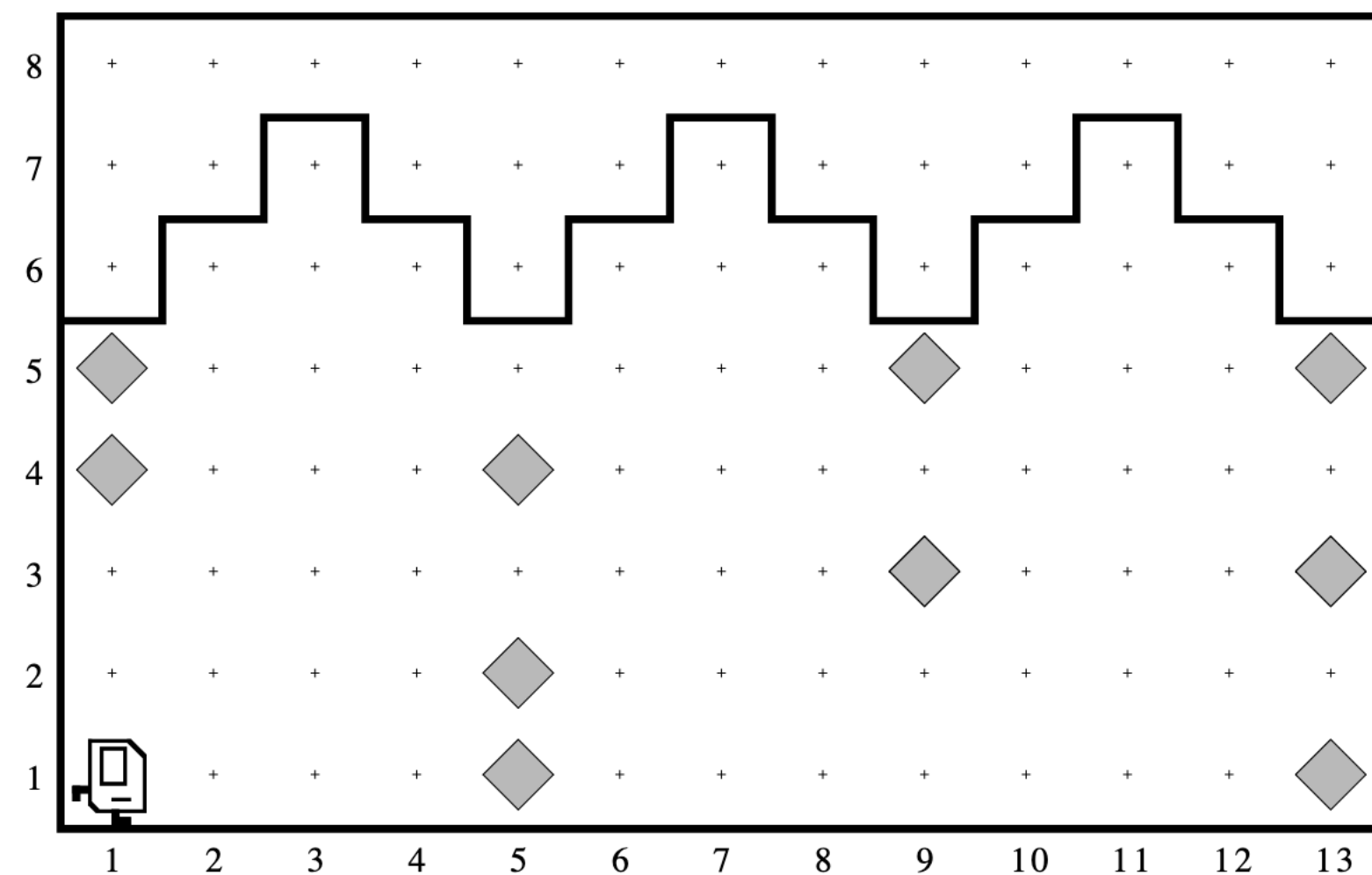
- If you can only look at a few lines of code, it's hard to tell (without broader context) whether that code is safe
- Getting broader context is impossible in the general case because of the halting problem
  - We can guesstimate what values get passed around in a program using dataflow analysis, and 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
- Is there some way we can make it easier to verify small snippets of code in isolation, without broader context?

Taking a step back

# What's the matter with `strncpy()`?

## Problem 2

Karel has been hired to repair the damage done to the Quad in the 1989 earthquake. In particular, Karel is to repair a set of arches where some of the stones (represented by beepers, of course) are missing from the columns supporting the arches, as follows:



- A natural decomposition for StoneMasonKarel is to implement a `repairColumn` function, then write:

```
while (frontIsClear()) {  
    repairColumn();  
    moveToNextColumn();  
}
```

```
repairColumn();
```

- This only works if `repairColumn` has a simple and consistent postcondition (e.g. Karel is facing east)
- If Karel is facing east sometimes but facing south other times, this program gets a lot more messy...

# What's the matter with strncpy()?

- Why do we care about specifying preconditions/postconditions?
  - If we can verify that preconditions/postconditions are upheld in isolation, then we can string together a bunch of components and simply verify that the preconditions/postconditions all fit together without needing to keep the *entire program* in our heads
- Postconditions should be **simple** and **consistently upheld**
  - If your postconditions are complicated, reasoning about function use and code correctness is really hard
  - It's the programmer's responsibility to make sure the postconditions are upheld (otherwise the function is broken). The compiler doesn't necessarily understand what your postconditions are

# Pre/postconditions should be *simple*

strncpy() man page:

*The strcpy() function copies the string pointed to by src, including the terminating null byte ('\0'), to the buffer pointed to by dest... 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. If the length of src is less than n, strncpy() writes additional null bytes to dest to ensure that a total of n bytes are written.*

Translation: strncpy returns a string *sometimes*. Other times, it returns a character buffer with n bytes of source data.

strncpy() man page:

*The strncpy() function copies up to size - 1 characters from the NUL-terminated string src to dst, NUL-terminating the result.*

Translation: strncpy returns a string.

# Pre/postconditions should be *consistently upheld*

- If your postcondition says your function returns something, you need to ensure it actually returns that thing in all cases
- Anyone using the function must ensure that they can handle whatever is returned
- `strncpy()` postcondition: returns a string *or* character array
- Anyone calling `strncpy()` must be prepared to handle a string *or* character array



# Pre/postconditions should be *consistently upheld*

- strncpy() postcondition: returns a string *or* character array
- Anyone calling strncpy() must be prepared to handle a string *or* character array
- It's up to the programmer to make sure to get this right. Is there a way we can get a compiler/static analyzer to check this?
- Key point: *compiler does not know what your postconditions are, **because it's not possible to express in the C language***

# Type systems

- The *types* of a programming language are the *nouns* of a spoken language
  - When you talk, what do you talk *about*?
- C type system: numbers, pointers, structs... not much else
  - Extremely simple: can learn most of the C language in half a quarter of CS 107
  - Simple != easy

strncpy definition:

```
strncpy(char *dst, const char *src,  
        size_t n);
```

*Takes a mutable char pointer, an  
immutable char pointer, and a number*

strncpy usage:

```
char buf[16];  
strncpy(buf, argv[1], sizeof(buf));  
printf("%s\n", buf);
```

*Passes a mutable char pointer (buf) ✓*

*Passes an immutable char pointer (argv[1]) ✓*

*Passes a number (sizeof(buf)) ✓*

- The pre/postconditions may be written in comments, but they are not present in the actual code, because the C language does not have a mechanism for them to be expressed
- Consequently: the compiler is unaware of what you're trying to do

\* A static analyzer may be able to *figure out* strncpy's behavior from looking at the code, but there is no way to figure out *everything*

# Type systems

- The *types* of a programming language are the *nouns* of a spoken language
  - When you talk, what do you talk *about*?
- C type system: numbers, pointers, structs... not much else
  - Extremely simple: can learn most of the C language in half a quarter of CS 107
  - Simple != easy

strchr definition:

```
char *strchr(const char *s, int c);
```

*Takes an immutable char pointer and a character(/number??) to look for*

*Returns a char pointer*

strchr usage:

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

*Passes a char\** ✓

*Passes a char* ✓

*Receives a char\** ✓, *prints it* 100

- The pre/postconditions may be written in comments, but they are not present in the actual code, because the C language does not have a mechanism for them to be expressed
- Consequently: the compiler is unaware of what you're trying to do

\* A static analyzer may be able to *figure out* that strchr might return NULL from looking at the code, but there is no way to figure out *everything*

Are there better type systems that we can use to specify  
our preconditions/postconditions *in the code*?

(implication: if the compiler can understand your pre/postconditions, it can verify that  
they are met)

Topic of next week! (Meet 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 a language with a better type system, 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