L07 Program Analysis

Static Analysis and Dynamic Analysis

Dr. Marco Elver
Systems Research Group
https://dse.in.tum.de/

Today's learning goals

- **Part I:** Faults and failures in software
  - Terminology and impact
- **Part II:** Program analysis trade-offs
  - Soundness, completeness, static vs. dynamic analysis
- **Part III:** Static analysis tools
  - Compiler warnings
  - Infer
  - SpotBugs
  - Clang Analyzer and Clang Tidy
- **Part IV:** Brief introduction to C
- **Part V:** Dynamic analysis tools
  - Undefined behavior
  - Dynamic binary instrumentation
  - Compiler-assisted instrumentation
  - Valgrind
  - Sanitizers
  - Program hardening

Outline

- **Part I: Faults and failures in software**
- **Part II:** Program analysis trade-offs
- **Part III:** Static analysis tools
- **Part IV:** Brief Introduction to C
- **Part V:** Dynamic analysis tools

Why program analysis?

Faults and failures in software

Fault-error-failure model [Randell 2000]:

1. **Human error (viz. programming error):** Human behavior that results in the introduction of **faults into a system (2)**.

2. **System fault ("bug"):** Characteristic of a software system that can lead to a **system error (3)**. Commonly referred to as system being "buggy".

3. **System error:** An erroneous system state during execution that can lead to unintended and unexpected behavior resulting in **system failure (4)**.

4. **System failure:** An event that occurs at some point in time when the system does not deliver a service as expected (e.g. crash, transmission of incorrect data, leaking sensitive data).

📖 I. Sommerville, "Reliability engineering" (Chapter 11), In "Software Engineering," 2015

Faults and failures in software

Not all faults cause errors, and not all errors cause failures:

1. *Code coverage:* Depending on program inputs, not all code in a program may be executed.

2. *Transient errors:* The program enters an invalid state only briefly, and there are no observable (unexpected) side-effects; this may happen when the program performs work that is aborted due to an interruption, retried, or otherwise reset.

3. *Fault detection or protection:* Erroneous behavior is discovered and corrected before it affects system services.

📖 I. Sommerville, “Reliability engineering” (Chapter 11), In “Software Engineering,” 2015

Avoiding faults and failures in software

Fault-avoidance: Software design and implementation process uses approaches to avoid *programming errors*, minimizing faults introduced.

📜 *Expressive type systems, advanced programming languages, formal methods*

Examples:

- Ada SPARK
- Agda
- Coq
- Isabelle
- Haskell
- Rust

📖 I. Sommerville, “Reliability engineering” (Chapter 11), In “Software Engineering,” 2015

Avoiding faults and failures in software

Fault-detection: Verification and validation processes to discover and remove *program faults* before deploying to “production”.

🔍 *Dynamic & static program analysis* (detect faults in specific executions)

Examples (discussed in this lecture):

- Infer
- SpotBugs
- Clang-Tidy
- AddressSanitizer, MemorySanitizer, ThreadSanitizer, …
- Valgrind

📖 I. Sommerville, “Reliability engineering” (Chapter 11), In “Software Engineering,” 2015

Avoiding faults and failures in software

Fault-tolerance: System detects faults in specific executions at runtime, mitigated on detection.

🔍 *Low-cost dynamic program analysis* 🏛️ *Fault-tolerant software architecture*

Examples (discussed in this lecture):

- Lightweight bounds-checking
- Stack Canaries
- Stack and Heap Initialization
- Hardened Memory Allocator
- Memory Tagging

📖 I. Sommerville, “Reliability engineering” (Chapter 11), In “Software Engineering,” 2015

Types of faults and errors

Undefined behavior

- Programming language specification (e.g. "C++ Standard") defines PL semantics: a subset of syntactically valid programs may contain faults resulting in "undefined behavior" errors.

🔥 *Anything allowed: crash, catch fire, eat your lunch…*

- Not all sources of undefined behavior can be detected by the compiler.
- **Examples:** null-deref, data races, uses of uninitialized memory, use-after-free accesses, out-of-bounds accesses, stack use-after-return.

🔍 ***Can be found with dynamic and static analysis***

Types of faults and errors

Semantic faults

- Faults that don't cause "undefined behavior", but still result in system errors.
- System deviates from its intended behavior.
- *Who defines intended behavior?*
  - Formal specification, reference implementation, documentation, manual
  - *Worst case:* not written down, but in programmer's head

📜 *Can be found with machine-checkable specifications*

📖 I. Sommerville, "Formal methods and dependability" (Chapter 10.5), In "Software Engineering," 2015

Outline

- ~~**Part I:** Faults and failures in software~~
- **Part II:** **Program analysis trade-offs**
  - Soundness
  - Completeness
  - Static vs. dynamic analysis
- **Part III:** Static analysis tools
- **Part IV:** Brief introduction to C
- **Part V:** Dynamic analysis tools

Program analysis is essential for software quality

1. Too costly to audit large software systems manually.
2. Program analysis techniques required for automated analysis.
3. Enables faster feature development and rollout.
4. To pick appropriate program analysis must understand desired system *availability and reliability* along with cost trade-offs.

Reliability: probability of failure-free operation over a specified time

Availability: probability that a system will be operational and deliver its services

📖 I. Sommerville, "Reliability engineering" (Chapter 11), In "Software Engineering," 2015

Trade-offs in program analysis

Soundness: if property P is provable, then P is true

Completeness: if property P is true, then P is provable

	Complete analysis	**Incomplete analysis**
Sound analysis	No false positives nor false negatives! *Fundamentally hard to achieve* (“Gödel's incompleteness theorem”).	May emit false positives, but has no false negatives: *if analysis says program is error-free, then it really is.*
Unsound analysis	May have false negatives, but does not emit false positives: *if analysis says there is an error, it’s a real error; if analysis says there are no errors, there may still be uncaught errors.*	May emit false positives and have false negatives: *every report needs to be scrutinized, absence of reports does not imply error-freedom (can result in bad developer experience).*

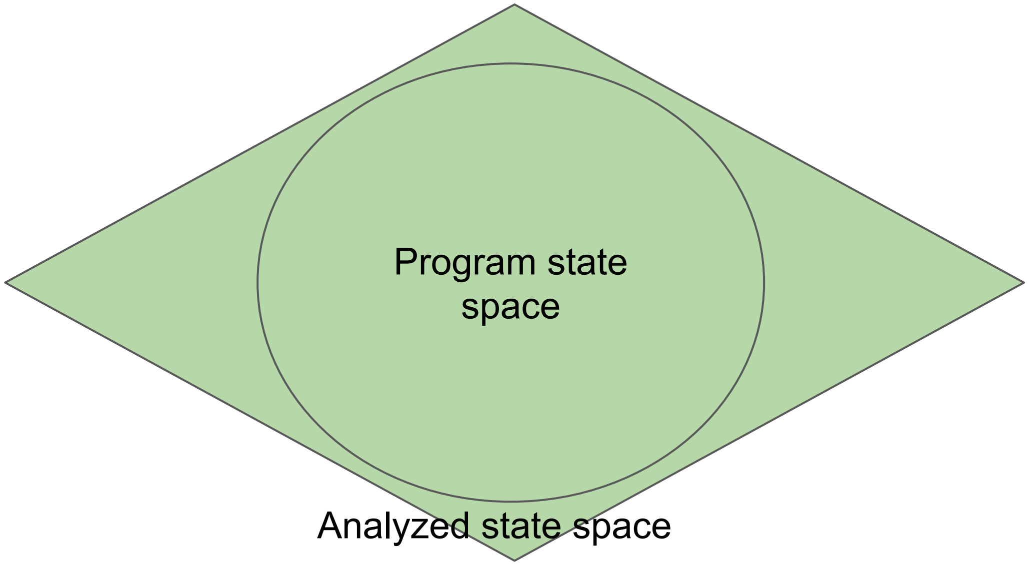
Static Analysis

- *Static program analysis* is about analyzing a piece of code “statically”: the analysis only inspects the source code without executing or running it.
- Static analysis reports can point out *system faults*, *errors*, or resulting *failures*
  - *Tools are often designed to allow for high quality diagnosis.*

📖 H. Adkins et al., “Static Program Analysis” (Chapter 13), In “Building Secure & Reliable Systems,” 2020

Static Analysis

- Can cover large sets of states very quickly and cheaply.
- *Completeness often traded against unsoundness:* tools prefer to produce useful signals (true positives) while keeping noise (false positives) low.



📖 H. Adkins et al., “Static Program Analysis” (Chapter 13), In “Building Secure & Reliable Systems,” 2020

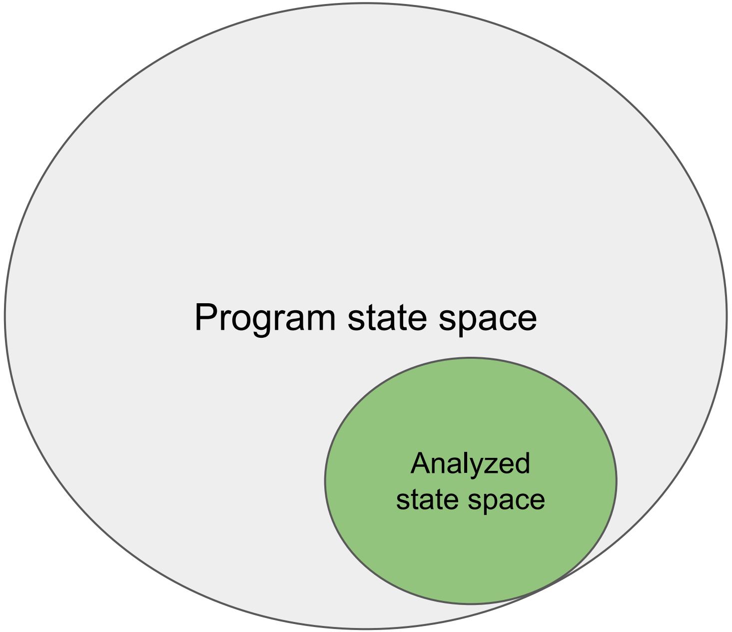
Dynamic Analysis

- *Dynamic program analysis* is about analyzing a piece of code "dynamically": the analysis observes the program as it is being executed.
- Dynamic analysis reports typically point out *system errors* or *failures.*
  - Can rarely deduce the underlying *system fault.*
  - Quality of diagnostics often inversely correlated with the performance of a tool.

📖 H. Adkins et al., "Dynamic Program Analysis" (Chapter 13), In "Building Secure & Reliable Systems," 2020

Dynamic Analysis

- Only the state space that was *covered* during execution is analyzed.
- If covered state space is non-exhaustive, the analysis will always be *unsound*.

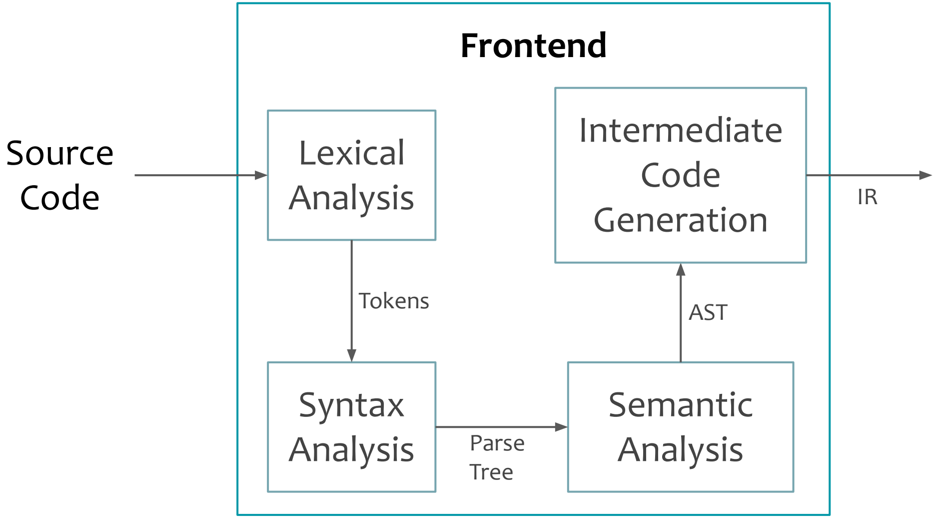


📖 H. Adkins et al., "Dynamic Program Analysis" (Chapter 13), In "Building Secure & Reliable Systems," 2020

Outline

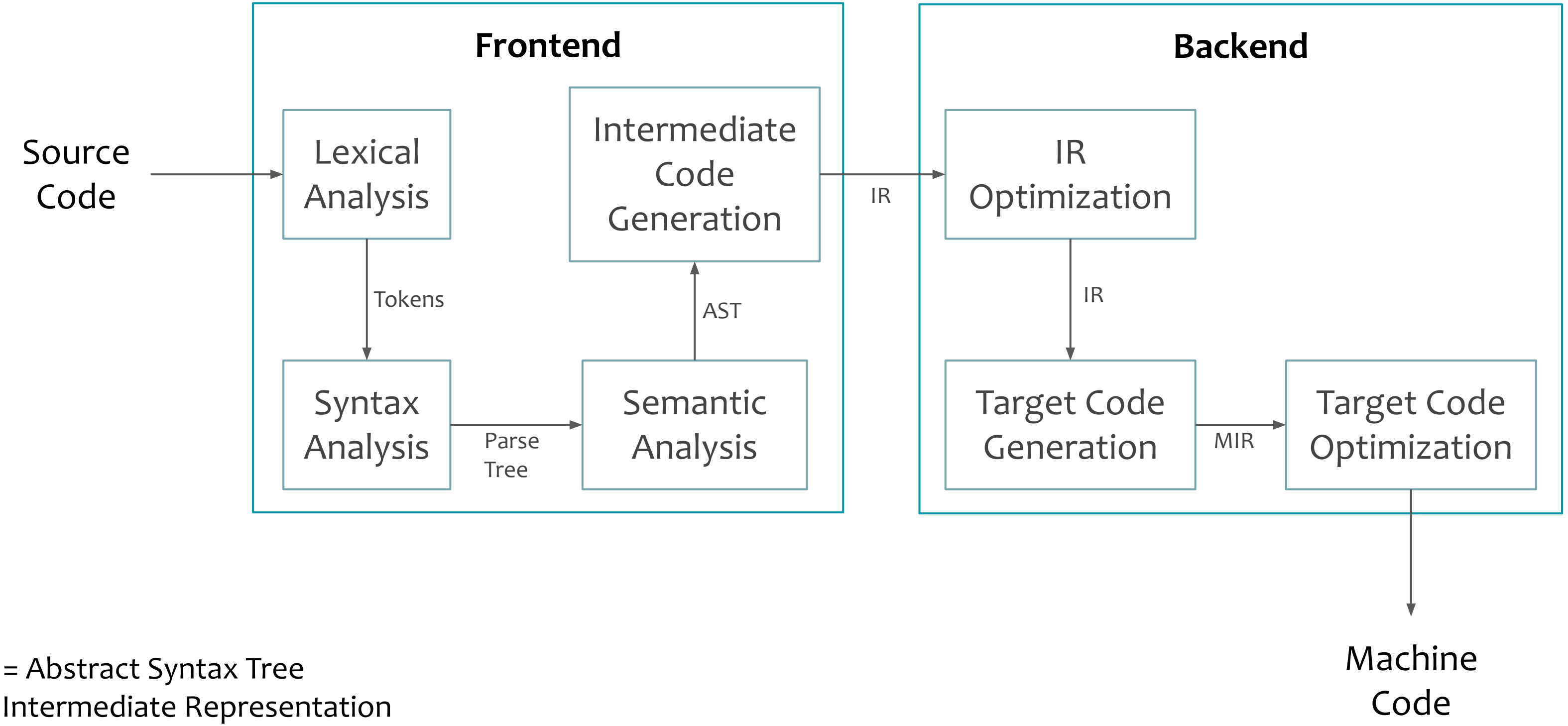
- ~~**Part I:** Faults and failures in software~~
- ~~**Part II:** Program analysis trade-offs~~
- **Part III: Static analysis tools**
    - Compiler warnings
    - Infer
    - SpotBugs
    - Clang Analyzer and Clang Tidy
- **Part IV:** Brief introduction to C
- **Part V:** Dynamic analysis tools

Compilation Pipeline



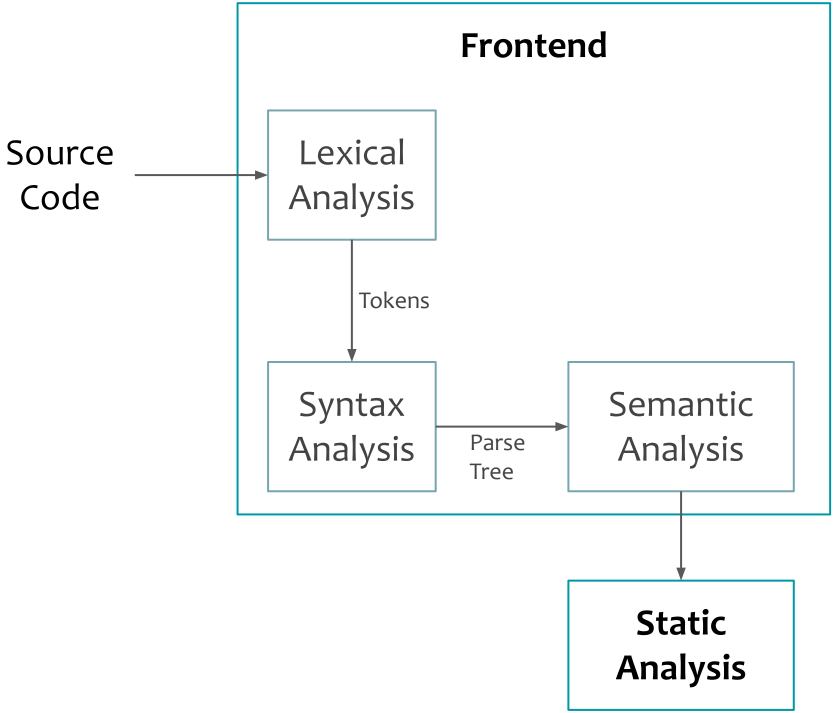
AST = Abstract Syntax Tree
IR = Intermediate Representation

Compilation Pipeline



AST = Abstract Syntax Tree
IR = Intermediate Representation
MIR = Machine Intermediate Representation

Compilation Pipeline



Static Analysis: Compiler errors, warnings & Linters

Built-in analyses by the language compiler and default toolchain:

- **Programming Languages:** many
- **False positives:** depends
- **False negatives:** depends
- **Cost:** very low

Availability:

- Java (more with -Xlint)
- C/C++ with GCC, Clang, MSVC (more with -Wall, -Wextra, or /Wall respectively)
- Rust (more with Clippy)
- Many many more – check your favorite language compiler…

Static Analysis: Compiler errors, warnings & Linters

```java
// Example.java
class Example {
  int foo(int x) {
      int y;
      if (x > 42)
          y = x * 123;
      return x + y;
  }
}
```

Static Analysis: Compiler errors, warnings & Linters

```java
// Example.java
class Example {
  int foo(int x) {
     int y;
     if (x > 42)
         y = x * 123;
     return x + y;
  }
}
```

```
$ javac Example.java
Example.java:6: error: variable y might not
have been initialized
     return x + y;
                ^
1 error
```

Static Analysis: Infer

- **Programming Languages:** Java, C, C++, Objective-C
- **False positives:** few
- **False negatives:** yes
- **Cost:** low

Description:

- Static analysis tool originally developed at Facebook.
- Released as open source software.
- Usage: `$> infer run -- <regular compile command>`
- Web: fbinfer.com

Static Analysis: Infer

Types of detected issues:

- Null Dereference
- Memory Leak
- Resource Leak
- Empty Vector Access
- Locking and Synchronization Issues
- Static Initialization Order Fiasco [C++]
- Premature nil-Termination Argument [C++]
- Performance Critical Calls to Expensive Method [Java]
- … many more ⇒ fbinfer.com/docs/all-issue-types

Static Analysis: Infer

How to install:

- Go to fbinfer.com/docs/getting-started/ and follow latest instructions.
  - Download from github.com/facebook/infer/releases/latest
  - Mac OS (brew): `$> brew install infer`

Static Analysis: Infer

```java
// Example.java
class Example {
  String foo(int i) {
    return i > 0 ? "Hello!" : null;
  }
  int bar(int x) {
    return foo(x).length();
  }
}
```

Static Analysis: Infer

```java
// Example.java
class Example {
  String foo(int i) {
    return i > 0 ? "Hello!" : null;
  }
  int bar(int x) {
    return foo(x).length();
  }
}
```

```
$ infer run -- javac Example.java
...
Found 1 issue

./Example.java:6: error: NULL_DEREFERENCE

  object returned by foo(x) could be null and is
  dereferenced at line 6

  4.       }
  5.       int bar(int x) {
  6. >       return foo(x).length();
  7.       }
  8.   }
  9.
```

Static Analysis: SpotBugs

- **Programming Languages:** Java
- **False positives:** few
- **False negatives:** yes
- **Cost:** low

Description:

- Static analysis tool working at JVM byte code level.
- Released as open source software.
- Usage: `$> spotbugs -textui myApp.jar`
- Web: spotbugs.github.io

Static Analysis: SpotBugs

Classes of detected issues:

- Bad Practice, Style Issues
- Correctness
- Code Vulnerabilities, Security Issues
- Locking and Synchronization Issues
- Performance Issues
- Internationalization Issues
- Details: spotbugs.readthedocs.io/en/latest/bugDescriptions.html

Static Analysis: SpotBugs

How to install:

- Go to spotbugs.readthedocs.io/en/latest/installing.html and follow latest instructions.
    - Download from github.com/spotbugs/spotbugs/releases/latest

Static Analysis: SpotBugs

```java
// Example.java
public class Example {
  boolean ready;
  void awaitReady()
   throws InterruptedException {
    synchronized (this) {
      while (!ready) {
        Thread.sleep(1);
      }
    }
  }
}
```

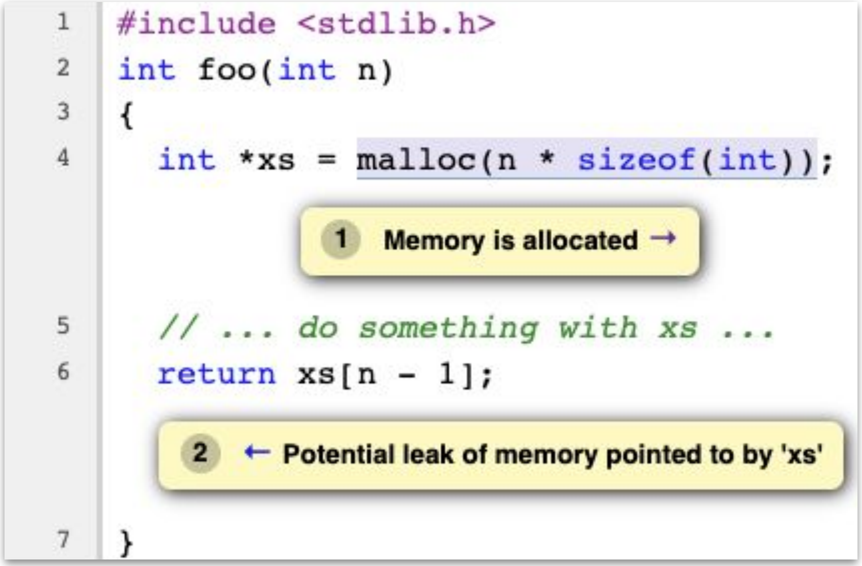
Static Analysis: SpotBugs

```java
// Example.java
public class Example {
  boolean ready;
  void awaitReady()
   throws InterruptedException {
    synchronized (this) {
      while (!ready) {
        Thread.sleep(1);
      }
    }
  }
}
```

```
# – Installing SpotBugs –
$ wget -q
https://github.com/spotbugs/spotbugs/releases/downl
oad/4.7.3/spotbugs-4.7.3.tgz
$ tar xzf spotbugs-4.7.3.tgz
$ chmod +x spotbugs-4.7.3/bin/spotbugs

# – Using SpotBugs –
$ javac Example.java
$ ./spotbugs-4.7.3/bin/spotbugs Example.class
M M SWL: Example.awaitReady() calls Thread.sleep()
with a lock held  At Example.java:[line 6]
```

Static Analysis: Clang Analyzer & Clang Tidy

- **Programming Languages:** C, C++, Objective-C
- **False positives:** few
- **False negatives:** yes
- **Cost:** low



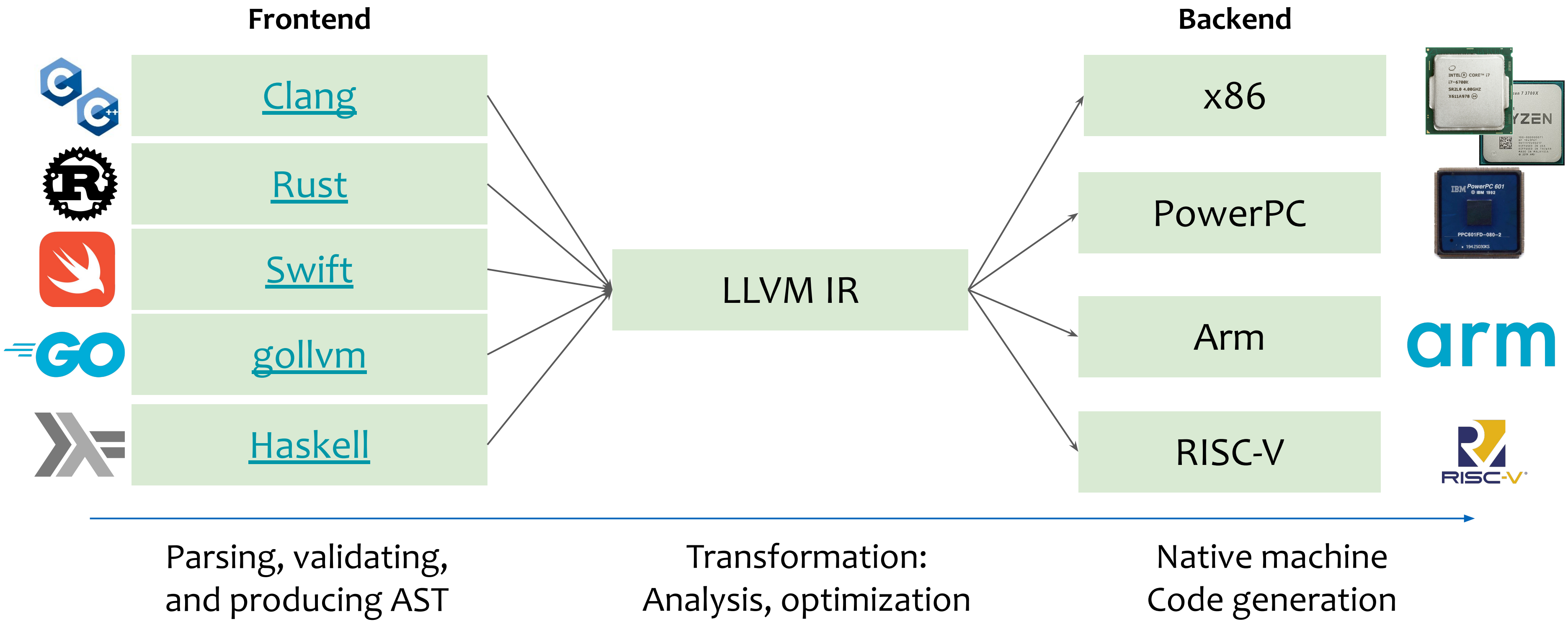
Description:

- Part of LLVM Compiler Infrastructure (open source).
- Provides framework to create custom analyses as plugins.
- Usage (clang-analyzer): `scan-build <regular compile command>`
- See clang-analyzer.llvm.org and clang.llvm.org/extra/clang-tidy for details.

LLVM Compiler Infrastructure

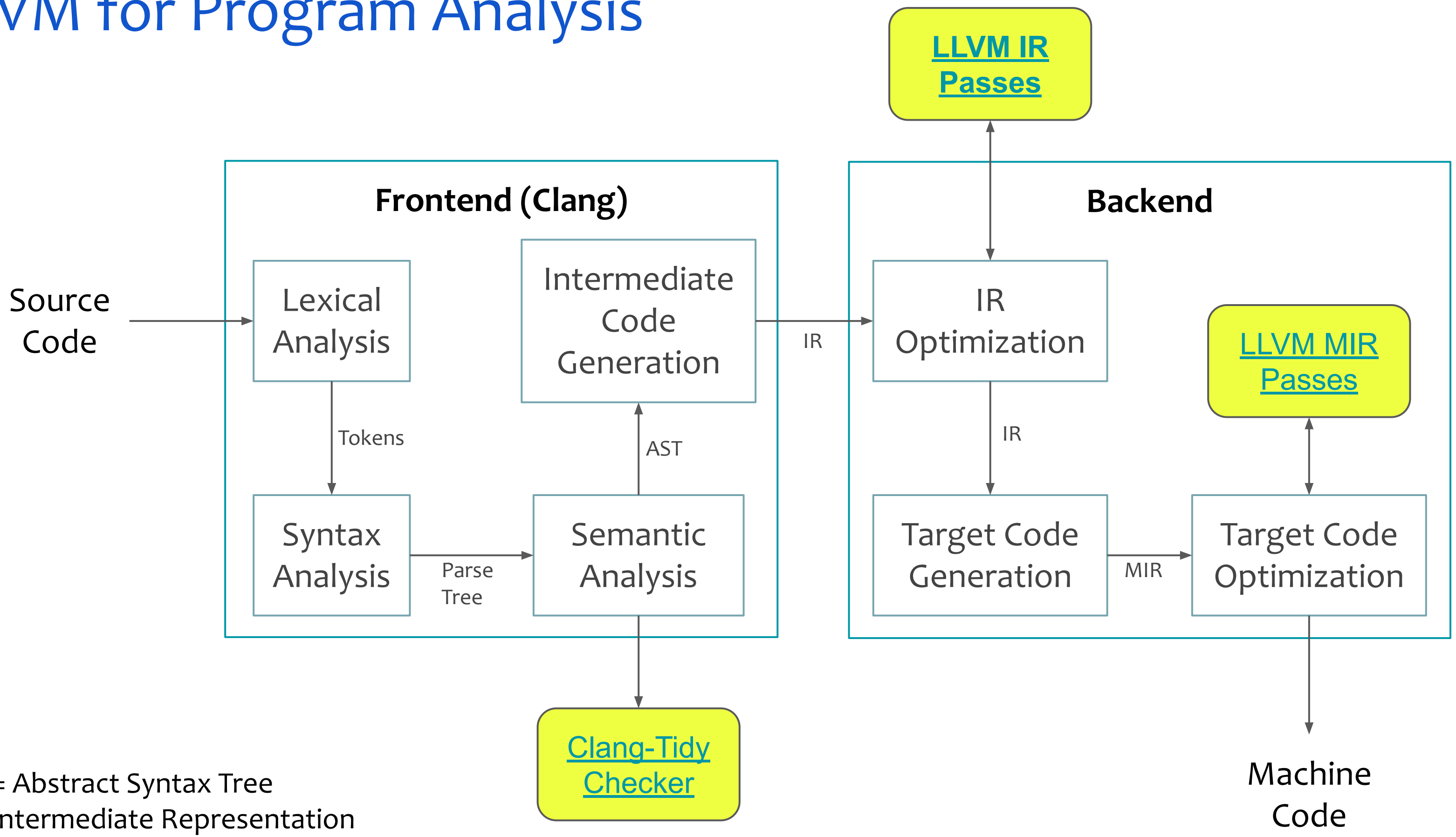
- "[...] collection of modular and reusable compiler and toolchain technologies"
- Started at UIUC in early 2000's, in response to GCC's monolithic architecture [Chris Lattner's PhD thesis]
- Modularity allows for a wide variety of uses across industries and in research!

LLVM Architecture: Three-Phase Design



Reference: https://aosabook.org/en/llvm.html

LLVM for Program Analysis



AST = Abstract Syntax Tree
IR = Intermediate Representation
MIR = Machine Intermediate Representation

References

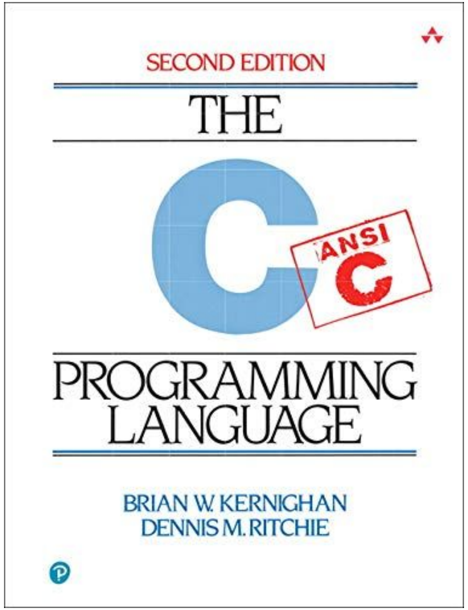
- javac manual
- GCC Warnings Options
- Infer
- SpotBugs
- Clang Analyzer
- Clang Tidy

Outline

- ~~**Part I:** Faults and failures in software~~
- ~~**Part II:** Program analysis trade-offs~~
- ~~**Part III:** Static analysis tools~~
- **Part IV: Brief introduction to C**
- **Part V:** Dynamic analysis tools

Brief Introduction to C

- C is everywhere – lowest common denominator!
- Developed in the 1970s alongside UNIX.
- Access to low-level operating system facilities and libraries via C API.
- Foreign Function Interfaces (FFI) in terms of a C ABI.
- Many newer languages derived from C syntax.
- Latest version of C standard is C23.



ABI: Application Binary Interface – interface in terms of executed machine code (binary level)

API: Application Programming Interface – typically defines a source-level interface (programmer level)

Introduction to C: Hello World

```c
// hello_world.c

#include <stdio.h>  // provides printf

// A single-line comment
/*
 * A multi-line
 * comment...
 */

int main(int argc, char *argv[])
{
    printf("Hello World!\n");
    return 0;
}
```

```
$> cc -Wall -o hello_world hello_world.c
$>./hello_world
Hello World!
```

💡 It is good practice to only include the headers that you need. In large projects, unneeded header inclusions increase compile times.

💡 Most UNIX-like systems should have a program "cc", which is usually just a link to the default system C compiler. You may also try to use "gcc" or "clang" instead, depending on what you have installed.

Introduction to C: Primitive Types

Java type	C type
int	int
short	short
long	long
float	float
double	double
char	char
byte	#include <stdint.h> int8_t
boolean	#include <stdbool.h> bool

Introduction to C: Pointers

Pointer declaration	`type *ptr;`
Take address of variable	`type values;` `type *ptr = &values;`
Dereference pointer	`*ptr`
Dereference pointer with offset (access Nth element after pointer)	`ptr[N]` `*(ptr + N)`
Read-only pointer	`const type *ptr;`

Introduction to C: Arrays

Simple array	`int values[32];`
Array initializer (partial, rest zero)	`int values[32] = {1, 2, 3, 4};`
Array initializer (implicit size)	`int values[] = {1, 2, 3, 4};`
Array initializer (all zero)	`int values[32] = {};`
Multi-dimensional array	`int values[2][3]; // 2x3 matrix`
Pointer to array (array to pointer decay)	`int *ptr = values;`

Introduction to C: Arrays & Pointers

```c
// stack_arrays.c
#include <stdio.h>
#include <stdlib.h>  // provides atoi()

#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))

static void print_values(const int *values, size_t count)
{
    for (int i = 0; i < count; ++i)
        printf("value[%d] = %d, ", i, values[i]);
}
int main(int argc, char *argv[])
{
    if (argc < 2) return 1;
    const int mul = atoi(argv[1]);
    int values[32] = {};
    for (int i = 0; i < ARRAY_SIZE(values); ++i) {
        values[i] = i * mul;
        if (values[i] > 100)
            break;
    }
    print_values(values, ARRAY_SIZE(values));
    return 0;
}
```

Introduction to C: Arrays & Pointers

```c
// stack_arrays.c
#include <stdio.h>
#include <stdlib.h>  // provides atoi()

#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))

static void print_values(const int *values, size_t count)
{
    for (int i = 0; i < count; ++i)
        printf("value[%d] = %d, ", i, values[i]);
}
int main(int argc, char *argv[])
{
    if (argc < 2) return 1;
    const int mul = atoi(argv[1]);
    int values[32] = {};
    for (int i = 0; i < ARRAY_SIZE(values); ++i) {
        values[i] = i * mul;
        if (values[i] > 100)
            break;
    }
    print_values(values, ARRAY_SIZE(values));
    return 0;
}
```

```
$> cc -Wall -o stack_arrays stack_arrays.c
$> ./stack_arrays 5
value[0] = 0, value[1] = 5, value[2] = 10, value[3] =
15, value[4] = 20, value[5] = 25, value[6] = 30,
value[7] = 35, value[8] = 40, value[9] = 45,
value[10] = 50, value[11] = 55, value[12] = 60,
value[13] = 65, value[14] = 70, value[15] = 75,
value[16] = 80, value[17] = 85, value[18] = 90,
value[19] = 95, value[20] = 100, value[21] = 105,
value[22] = 0, value[23] = 0, value[24] = 0,
value[25] = 0, value[26] = 0, value[27] = 0,
value[28] = 0, value[29] = 0, value[30] = 0,
value[31] = 0,
```

💡 C has complex initialization rules. If in doubt, explicitly initialize variables!

💡 Declare functions and global variables `static` if they are “private” to the source file.

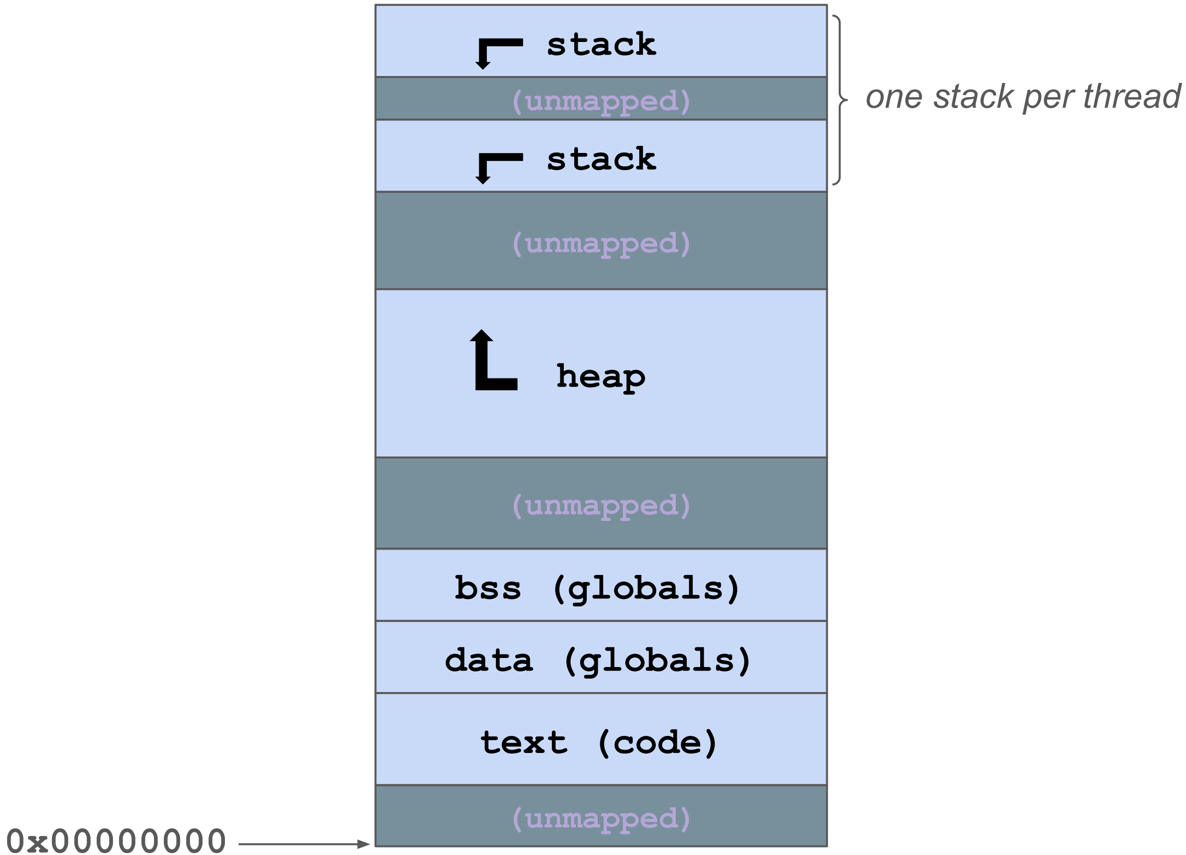
Introduction to C: Dynamic Memory Allocation

```c
// malloc.c

#include <stdlib.h>  // provides malloc(), free()

int main(int argc, char *argv[])
{
    int *values = (int *)malloc(32 * sizeof(int));
    // ... do something with values ...
    free(values);
    return 0;
}
```

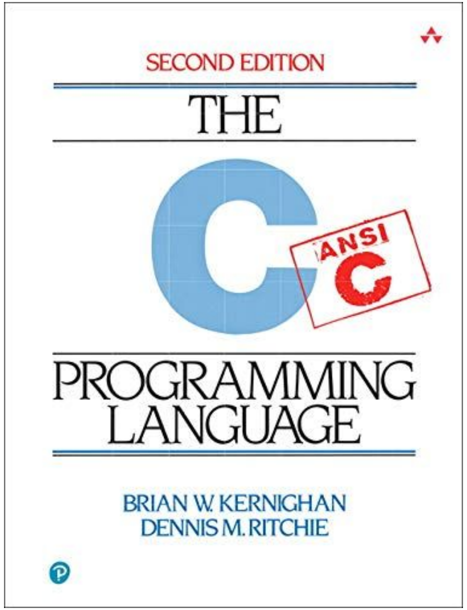
What is memory anyway?

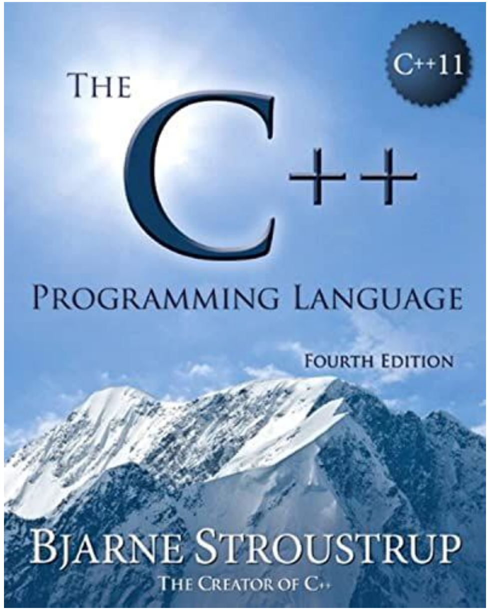
- Each process gets its own *virtual address space*
  - Virtual memory mapping to physical memory managed by OS kernel
  - Prohibits access to other processes' address spaces
- For each process, memory is allocated for:
  - Code
  - Globals
  - Stack (for function-local variables spilled from CPU registers)
  - Heap (dynamic memory allocation)
- Addresses may not always be the same
  - Affected by ASLR (more later)
  - Order of allocations (heap)

Process Memory Layout



References


SECOND EDITION
THE
C
ANSI C
PROGRAMMING LANGUAGE
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE


C++11
THE
C++
PROGRAMMING LANGUAGE
FOURTH EDITION
BJARNE STROUSTRUP
THE CREATOR OF C++

Outline

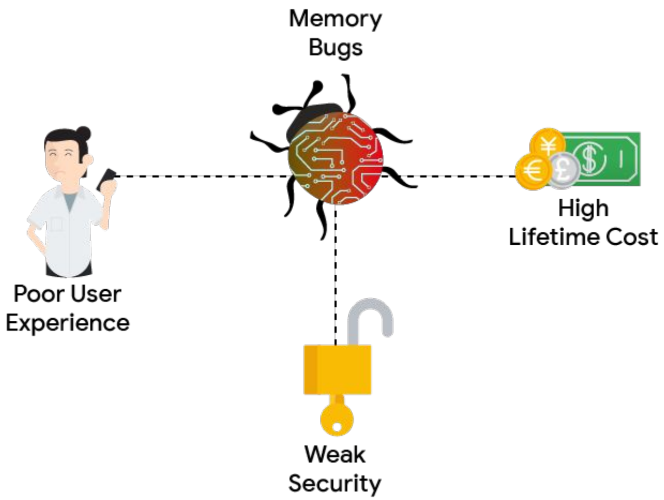
- ~~**Part I:** Faults and failures in software~~
- ~~**Part II:** Program analysis trade-offs~~
- ~~**Part III:** Static analysis tools~~
- ~~**Part IV:** Brief introduction to C~~
- **Part V: Dynamic analysis tools**
  - Undefined behavior
  - Dynamic binary instrumentation
  - Compiler-assisted instrumentation
  - Valgrind
  - Sanitizers
    - UndefinedBehaviorSanitizer
    - AddressSanitizer
    - ThreadSanitizer
    - MemorySanitizer
  - Program hardening

Undefined Behavior

Why "undefined behavior"?

- C and C++ were specifically designed for fine-grained control over low-level details, such as how memory is organized (**manual memory management**).
- *Unsafe languages* simply say: *some well-typed programs are* ***undefined*** 💥
    - Trade-off is simpler type system + higher performance (no dynamic error checking).
- *Safe languages* with manual memory management hard to design & implement.
    - Rust is considered safe in its "safe" subset.

Undefined Behaviour: Memory-Safety Errors



Memory-safety bugs *account for over 60% of high severity security vulnerabilities!*

📖 Source: source.android.com/docs/security/test/memory-safety

Memory Safety

Memory-safety error: An illegal access to unintended memory regions.

Two types of errors:

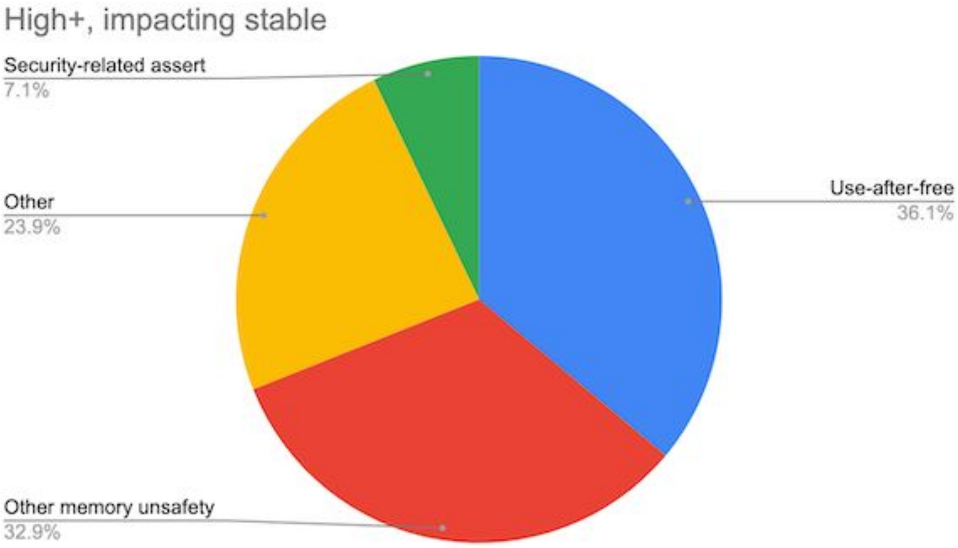
1. **Spatial errors:** unintended address
   - buffer overflow, stack overflow (out-of-bounds)
2. **Temporal errors:** unintended time
   - double free, dangling pointers (use-after-free, use-after-return)

Heartbleed in OpenSSL
(buffer overflow)

Memory Safety in Practice

Prevalent in almost all low-level C/C++ code:

- Example #1: Chromium project
  - 70% vulnerabilities are memory safety problems, and half of those are use-after-free bugs
- Example #2: Microsoft
  - 70% of vulnerabilities fixed in security patches are memory safety violations
- Example #3: Android
  - 75% vulnerabilities are memory safety issues



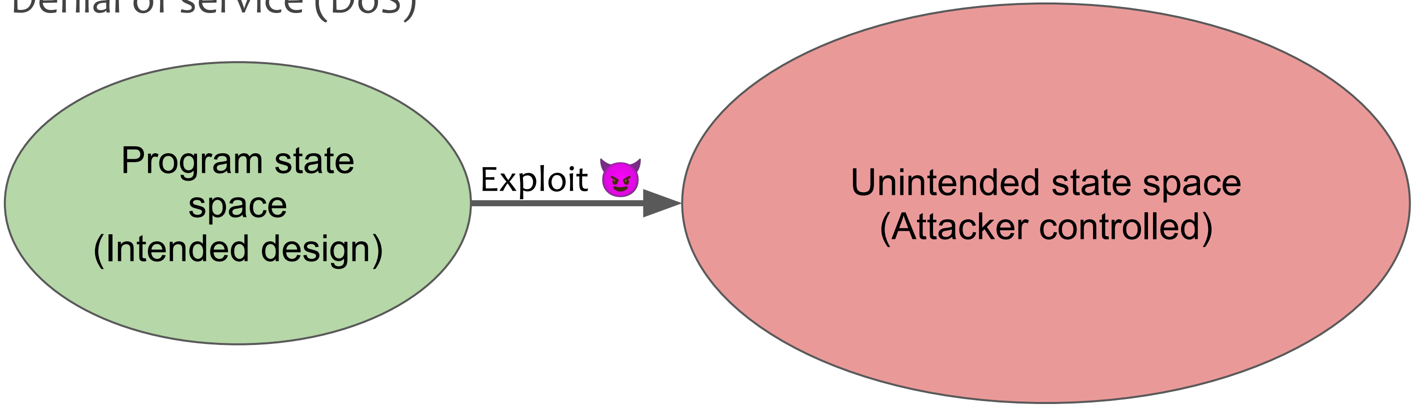
📖 Chromium project: https://www.chromium.org/Home/chromium-security/memory-safety
📖 Microsoft: https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/
📖 Android: https://security.googleblog.com/2019/05/queue-hardening-enhancements.html

Exploiting Memory Safety Errors

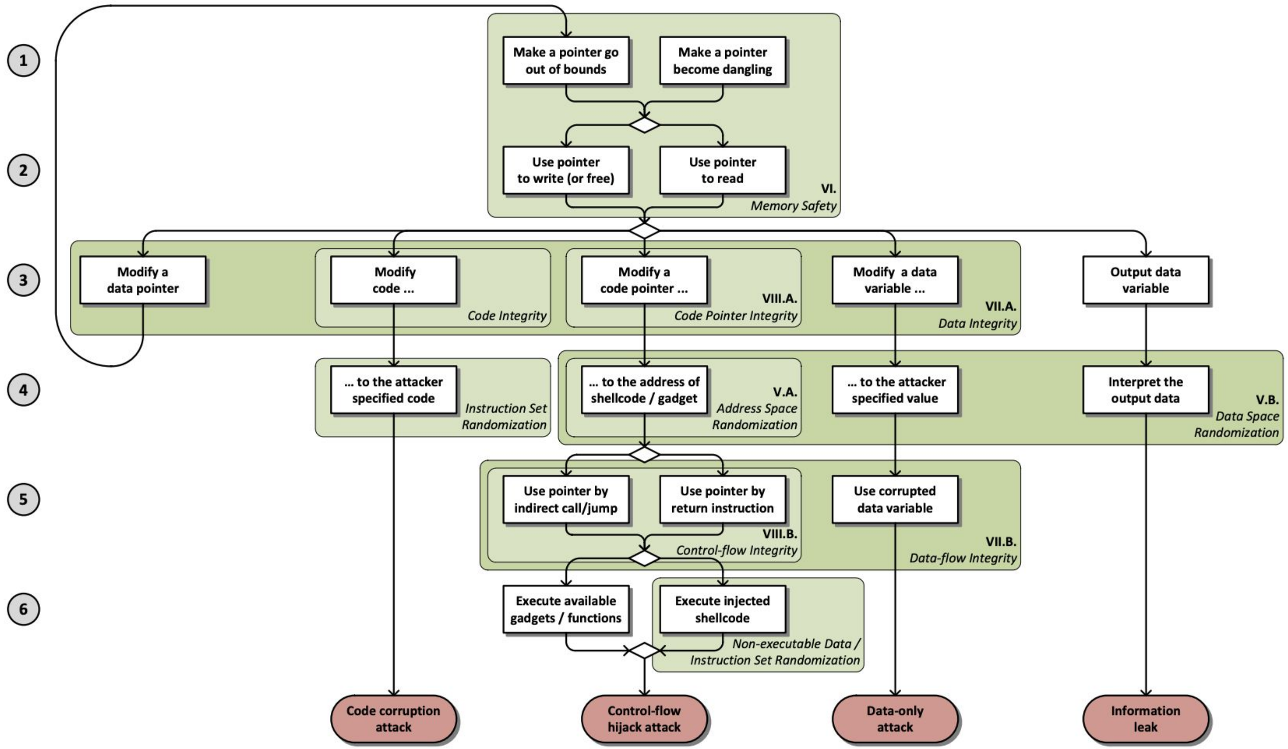
- Unmitigated errors where application keeps on running allow for entering unintended state ⇒ attacker takes control.

Binary Exploitation: taking control of application behavior in *unintended* ways.

  - Leak sensitive data (information leak)
  - Arbitrary code execution
  - Denial of service (DoS)



Exploiting Memory Safety Errors



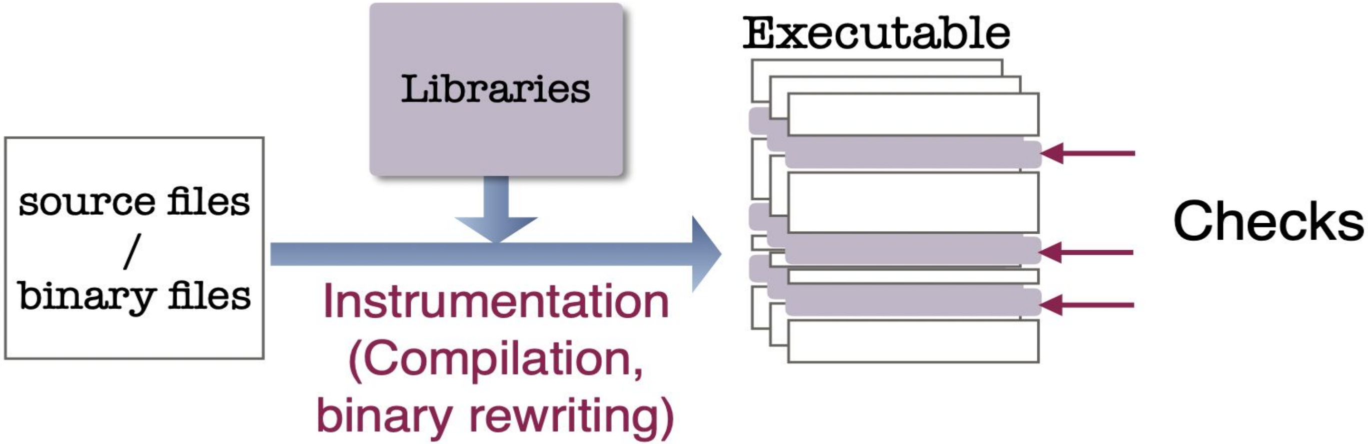
Memory-safety errors are the root cause of most security attacks [Szekeres et al. Oakland'13]

Undefined Behavior

- Integer overflow, shift out-of-bounds, null-deref, etc.
- Memory-safety errors:
  - Buffer overflow (out-of-bounds) accesses
  - Heap use-after-free
  - Stack use-after-return
  - Uses of uninitialized memory
  - Data races

How to catch them at runtime? Can we catch them cheaply?

Dynamic Analysis



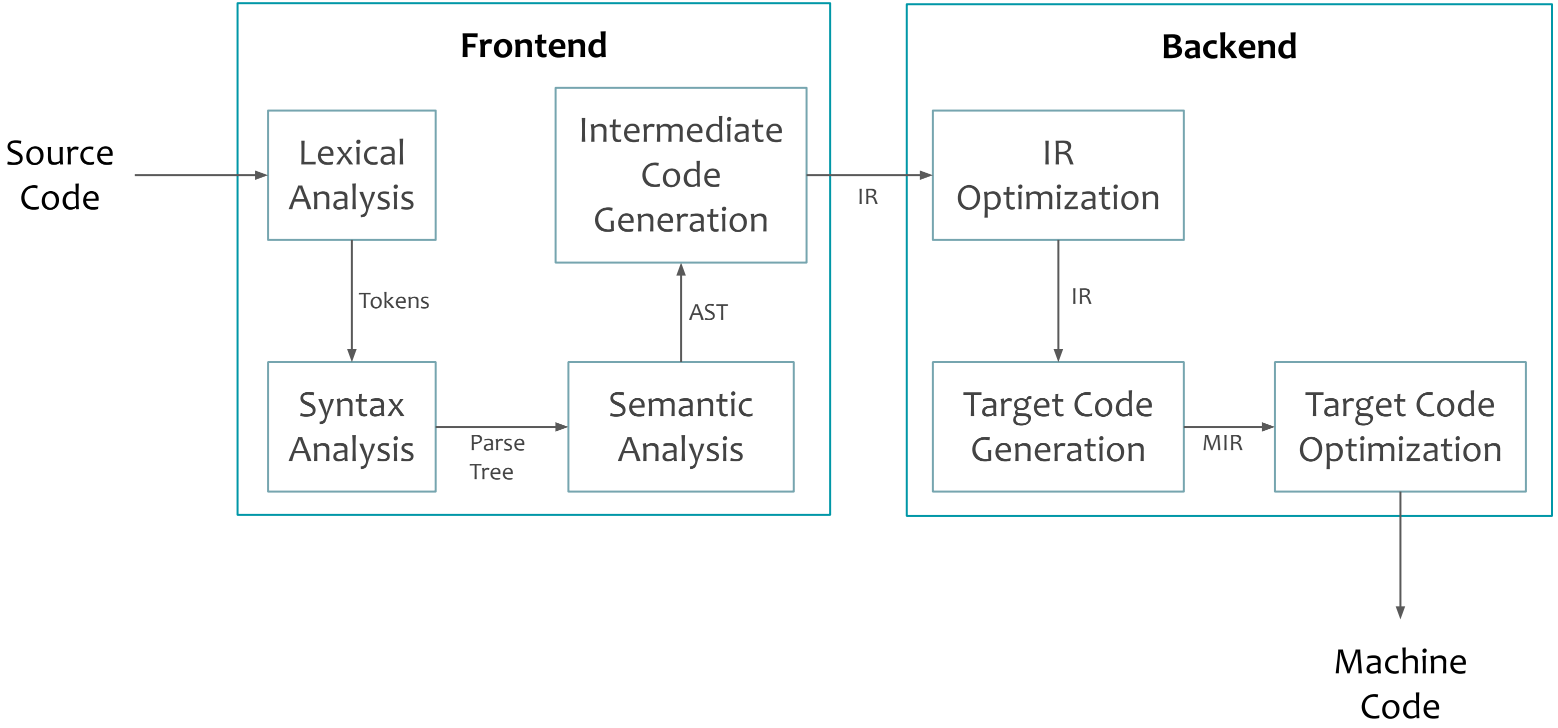
Dynamic Binary Instrumentation

- Instruments *unmodified* binary by inserting calls and/or emulating instructions.
- Usually results in very high runtime overheads.
- Unaware of source language semantics.
  - Analysis must be language-agnostic.
- Popular dynamic binary instrumentation frameworks:
  - Valgrind: valgrind.org
  - Intel Pin: www.intel.com/software/pintool
  - DynamoRIO: dynamorio.org
  - Can be used to build various dynamic program analysis.

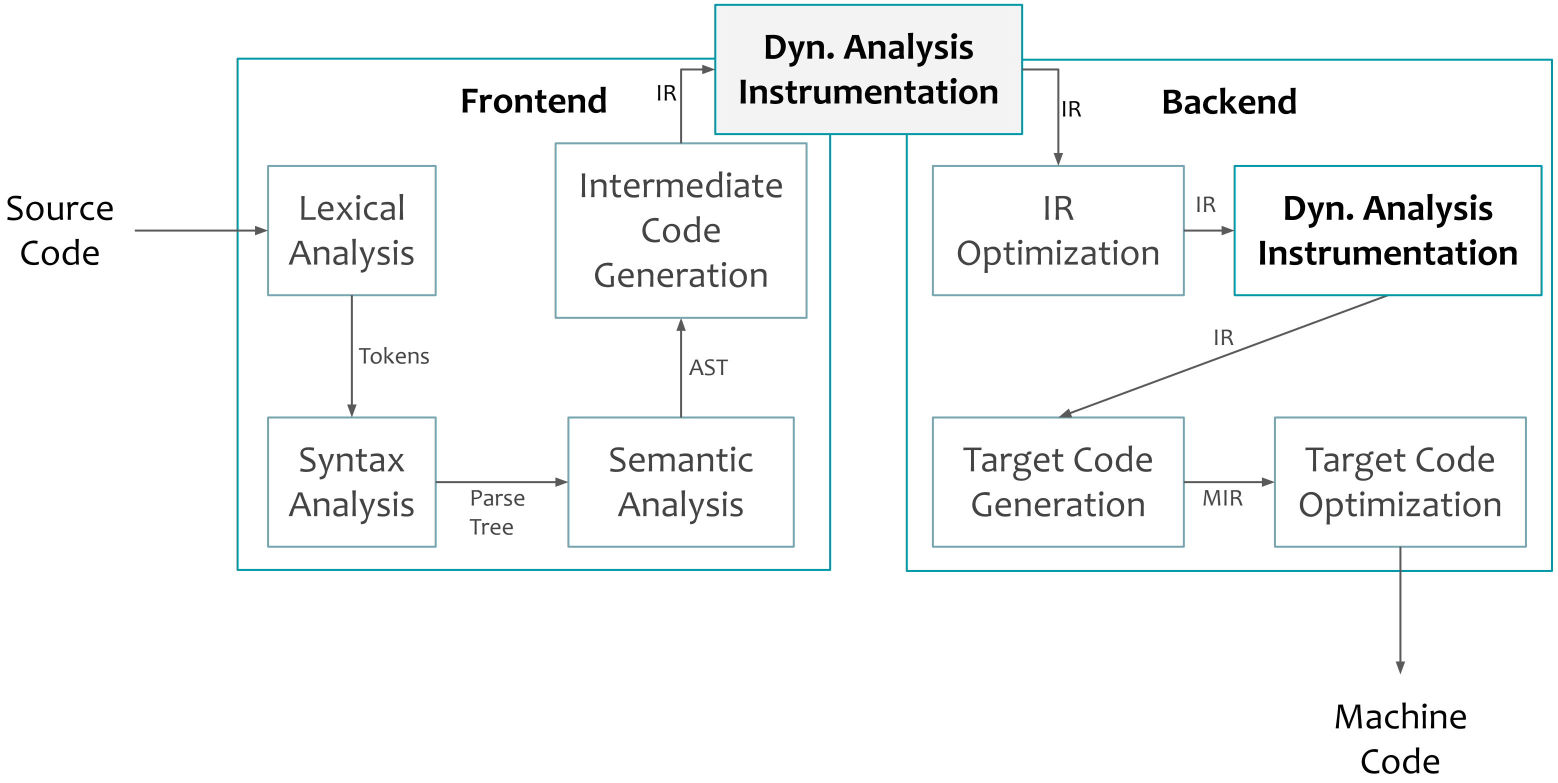
Compiler-Assisted Instrumentation

- Instruments *source compilation* by inserting instructions (calls, checks, etc.)
- Aware of source language semantics.
- Typically more performant than dynamic binary instrumentation.
    - No need to emulate instructions.
    - Compiler can optimize the sum of original code and instrumentation.

Compilation Pipeline



Compilation Pipeline



Undefined Behavior

- **Integer overflow, shift out-of-bounds, type-confusion, etc.**
- Memory-safety errors:
  - Buffer overflow (out-of-bounds) accesses
  - Heap use-after-free
  - Stack use-after-return
  - Uses of uninitialized memory
  - Data races

Undefined Behavior: Integer overflow

- C and C++ standards say that signed integer overflow is undefined.
- When C was invented (1970s), different CPUs behaved differently on int overflow.
  - Signed integers may be represented differently: 1s complement, 2s complement, sign-magnitude (these days all mainstream CPUs use 2s complement).
- **Note:** *Unsigned* integer overflow is defined.

```
int main(int argc, char *argv[])
{
    int k = 0x7ffffffe;
    k += argc;  // UB if argc >= 2
    return k;
}
```

Dynamic Analysis: UndefinedBehaviorSanitizer

- **Programming Languages:** C, C++
- **False positives:** no
- **False negatives:** uncovered error states
- **Cost:** low - moderate (depends on checks)

Description:

- *Compiler-instrumentation* based detector of various **C and C++ undefined behaviors.**
- Part of GCC and LLVM Compiler Infrastructure.
- Usage: `clang -fsanitize=undefined …`
- Usage: `gcc -fsanitize=undefined …`
- See clang.llvm.org/docs/UndefinedBehaviorSanitizer.html for details.

```
$ pygmentize ub.c
int main(int argc, char *argv[])
{
  int k = 0x7ffffffe;
  k += argc;
  return k;
}
$ clang -O -Wall -fsanitize=undefined ub.c
$ ./a.out
$ ./a.out a
ub.c:4:5: runtime error: signed integer overflow: 2147483646 + 2
 cannot be represented in type 'int'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior ub.c:4:5
```

```
$ pygmentize ub.c
int main(int argc, char *argv[])
{
  int k = 0x00ffffff;
  k <<= argc;
  return k;
}
$ clang -O -Wall -fsanitize=undefined ub.c
$ ./a.out 1
$ ./a.out 1 2
$ ./a.out 1 2 3
$ ./a.out 1 2 3 4
$ ./a.out 1 2 3 4 5
$ ./a.out 1 2 3 4 5 6
$ ./a.out 1 2 3 4 5 6 7
ub.c:4:5: runtime error: left shift of 16777215 by 8 places cann
ot be represented in type 'int'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior ub.c:4:5
```

Undefined Behavior

- ~~Integer overflow, shift out-of-bounds, type-confusion, etc.~~
- **Memory-safety errors:**
  - **Buffer overflow (out-of-bounds) accesses**
  - **Heap use-after-free**
  - **Stack use-after-return**
  - Uses of uninitialized memory
  - Data races

Undefined Behavior: Out-of-bounds accesses

- Accesses memory beyond the allocated memory.
    - No bounds checking by default.
    - Compiler may sometimes warn (if it can infer array size).
- May read random data, or corrupt other application state!
    - Can be exploited to leak memory, or control application in unintended ways!

```c
#include <ctype.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
    char buf[10];
    strcpy(buf, argv[1]);  // unchecked strcpy!
    for (char *c = buf; *c; ++c)
        *c = toupper(*c);
    printf("%s\n", buf);
    return 0;
 }
```

Undefined Behavior: Heap use-after-free

- Accesses recently unallocated heap memory.
  - Memory may already have been recycled, or given back to OS.
- May read random data, or corrupt other application state!
  - Can be exploited to leak memory, or control application in unintended ways!

```c
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
    char *buf = (char *)malloc(strlen(argv[1]));
    strcpy(buf, argv[1]);
    for (char *c = buf; *c; ++c)
        *c = toupper(*c);
    free(buf);             // too early!
    printf("%s\n", buf);  // use-after-free!
    return 0;
}
```

Undefined Behavior: Stack use-after-return

- Access to memory in invalid stack frame.
  - Stack memory may already have been reused in the next call.
- May read random data, or corrupt other application state!
  - Can be exploited to leak memory, or control application in unintended ways!

```c
#include <ctype.h>
#include <stdio.h>
#include <string.h>
static const char *strtoupper(const char *str)
{
    char buf[64];
    strcpy(buf, str);
    for (char *c = buf; *c; ++c)
        *c = toupper(*c);
    return buf;  // return of pointer to stack var!
}
```

```c
int main(int argc, char *argv[])
{
    const char *buf = strtoupper(argv[1]);
    printf("%s\n", buf);  // use-after-return!
    return 0;
}
```

Dynamic Analysis: Valgrind

- **Programming Languages:** native compiled (C, C++, …)
- **False positives:** no
- **False negatives:** yes
- **Cost:** high

Description:

- *Binary instrumentation* framework for detecting **memory-safety and concurrency bugs**.
- Released as open source software.
- Works on unmodified binaries by *simulating* instructions and analyzing instructions during execution.
- Usage: `valgrind --tool=<name> <binary>`
- See valgrind.org for details.

```
$ pygmentize bug.c
#include <ctype.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char buf[10];
  strcpy(buf, argv[1]);
  for (char *c = buf; *c; ++c)
    *c = toupper(*c);
  printf("%s\n", buf);
  return 0;
}
$ gcc -Wall bug.c
$ ./a.out abcd
ABCD
$ valgrind -q ./a.out helloworldhelloworld1234567890
HELLOWORLD????
==910313== Jump to the invalid address stated on the next line
==910313==    at 0x30393837: ???
==910313==  Address 0x30393837 is not stack'd, malloc'd or (recently) free'd
==910313==
==910313==
==910313== Process terminating with default action of signal 11 (SIGSEGV)
==910313==  Access not within mapped region at address 0x30393837
==910313==    at 0x30393837: ???
==910313==  If you believe this happened as a result of a stack
==910313==  overflow in your program's main thread (unlikely but
==910313==  possible), you can try to increase the size of the
==910313==  main thread stack using the --main-stacksize= flag.
==910313==  The main thread stack size used in this run was 8388608.
Segmentation fault
```

Dynamic Analysis: AddressSanitizer

- **Programming Languages:** C, C++, (Rust)
- **False positives:** no
- **False negatives:** uncovered error states
- **Cost:** moderate

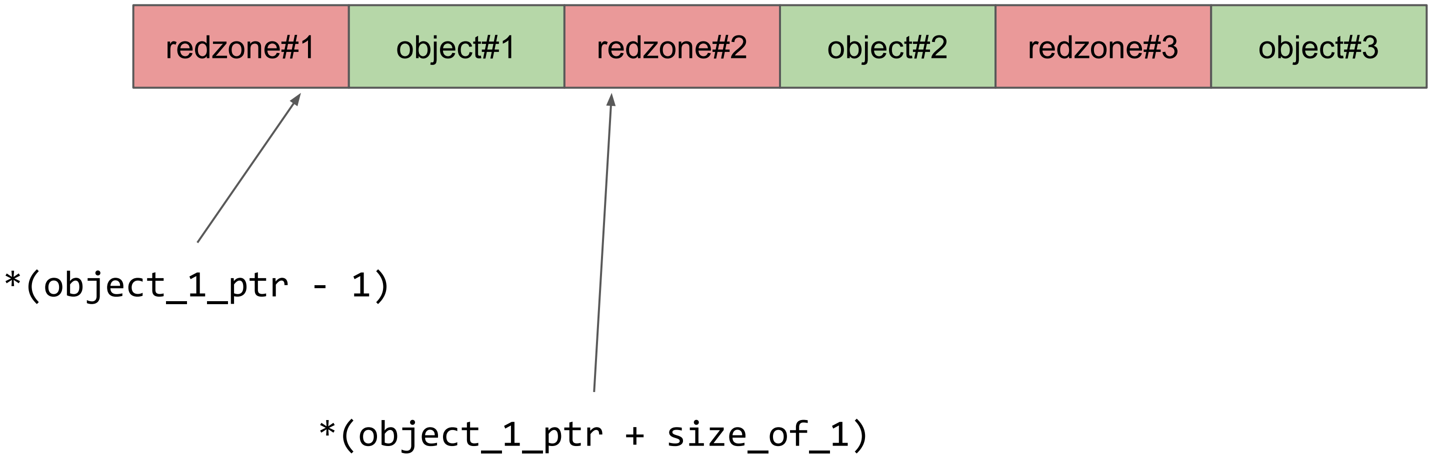
Description:

- *Compiler-instrumentation* based **memory-safety error detector**
  - **out-of-bounds, use-after-free, use-after-return**
- Part of GCC and LLVM Compiler Infrastructure.
- Usage: `clang -fsanitize=address …`
- Usage: `gcc -fsanitize=address …`
- See clang.llvm.org/docs/AddressSanitizer.html for details.

```
$ pygmentize bug.c
#include <ctype.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char buf[10];
  strcpy(buf, argv[1]);
  for (char *c = buf; *c; ++c)
    *c = toupper(*c);
  printf("%s\n", buf);
  return 0;
}
$ clang -fsanitize=address bug.c
$ ./a.out abcd
ABCD
$ ./a.out helloworldhelloworld1234567890
=================================================================
==910604==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fff148b050a
 at pc 0x55bf1fd3f8f7 bp 0x7fff148b04d0 sp 0x7fff148afc98
WRITE of size 31 at 0x7fff148b050a thread T0
    #0 0x55bf1fd3f8f6 in strcpy (/tmp/buggy/a.out+0x8e8f6) (BuildId: 384723e2c5454
2ad0ff1340eac2c0235e064e49e)
    #1 0x55bf1fd8ffc1 in main (/tmp/buggy/a.out+0xdefc1) (BuildId: 384723e2c54542a
d0ff1340eac2c0235e064e49e)
    #2 0x7f5fe5db3189 in __libc_start_call_main csu/../sysdeps/nptl/libc_start_cal
l_main.h:58:16
    #3 0x7f5fe5db3244 in __libc_start_main csu/../csu/libc-start.c:381:3
    #4 0x55bf1fcd2310 in _start (/tmp/buggy/a.out+0x21310) (BuildId: 384723e2c5454
2ad0ff1340eac2c0235e064e49e)

Address 0x7fff148b050a is located in stack of thread T0 at offset 42 in frame
    #0 0x55bf1fd8febf in main (/tmp/buggy/a.out+0xdeebf) (BuildId: 384723e2c54542a
d0ff1340eac2c0235e064e49e)

  This frame has 1 object(s):
    [32, 42) 'buf' <== Memory access at offset 42 overflows this variable
```

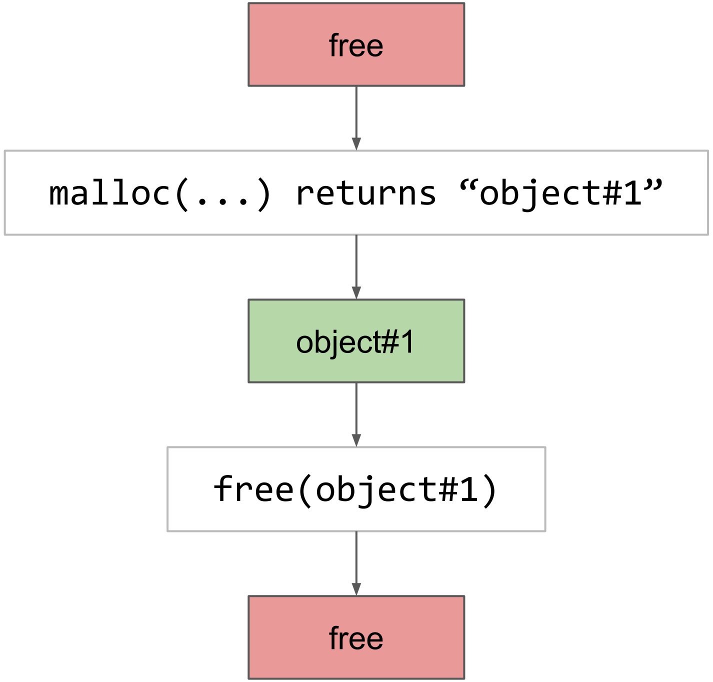
Dynamic Analysis: AddressSanitizer

Detects out-of-bounds accesses by adding “redzones” (or “trip wires”)



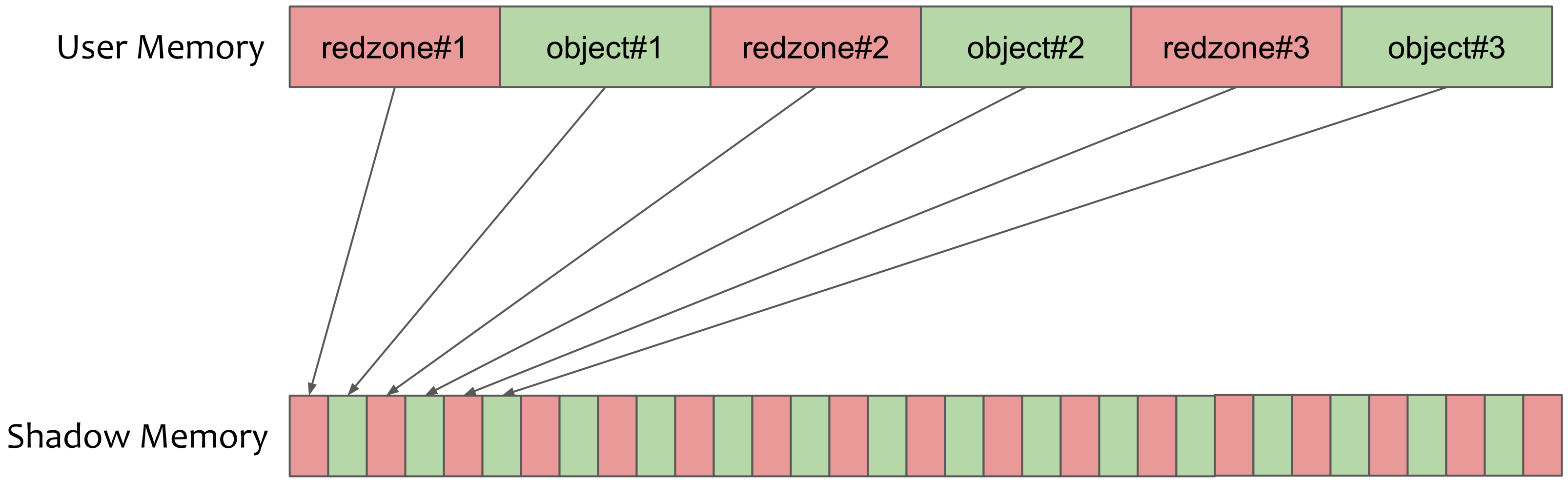
Dynamic Analysis: AddressSanitizer

Detects use-after-free and use-after-return by “poisoning” memory on free



Dynamic Analysis: AddressSanitizer

Each addressable byte has associated metadata ("shadow memory")



Undefined Behavior

- ~~Integer overflow, shift out-of-bounds, type-confusion, etc.~~
- **Memory-safety errors:**
  - ~~Buffer overflow (out-of-bounds) accesses~~
  - ~~Heap use-after-free~~
  - ~~Stack use-after-return~~
  - **Uses of uninitialized memory**
  - Data races

Undefined Behavior: Uses of uninitialized memory

- Access memory that has not been initialized.
  - C and C++ have complex rules for when some memory is initialized.
- May read random data or even old data from recycled memory!
  - Could be exploited to leak sensitive data!

```c
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
    char buf[10];
    strcpy(buf, argv[1]);
    if (buf[3] == 'x')
        printf("hello world\n");
    return 0;
}
```

Dynamic Analysis: MemorySanitizer

- **Programming Languages:** C, C++, (Rust)
- **False positives:** no
- **False negatives:** uncovered error states
- **Cost:** moderate

Description:

- *Compiler-instrumentation* based **use-of-uninitialized memory detector**.
- Part of GCC and LLVM Compiler Infrastructure.
- Usage: `clang -fsanitize=memory …`
  - Unavailable in GCC
- See clang.llvm.org/docs/MemorySanitizer.html for details.

```
$ pygmentize uninit.c
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char buf[10];
  strcpy(buf, argv[1]);
  if (buf[3] == 'x')
    printf("hello world\n");
  return 0;
}
$ clang -O -Wall -fsanitize=memory uninit.c
$ ./a.out 0123
$ ./a.out 012x
hello world
$ ./a.out 01
==924612==WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x56011236e35f in main (/tmp/buggy/a.out+0xa735f) (BuildId
    #1 0x7f10e45ca189 in __libc_start_call_main csu/../sysdeps/np
    #2 0x7f10e45ca244 in __libc_start_main csu/../csu/libc-start.
    #3 0x5601122e82a0 in _start (/tmp/buggy/a.out+0x212a0) (Build

SUMMARY: MemorySanitizer: use-of-uninitialized-value (/tmp/buggy/
Exiting
```

Undefined Behavior

- ~~Integer overflow, shift out-of-bounds, type-confusion, etc.~~
- **Memory-safety errors:**
  - ~~Buffer overflow (out-of-bounds) accesses~~
  - ~~Heap use-after-free~~
  - ~~Stack use-after-return~~
  - ~~Uses of uninitialized memory~~
  - **Data races**

Undefined Behavior: Data Races

- Data races occur if **two conflicting accesses execute concurrently**.
  - Accesses conflict if they access the *same memory location*, at least *one is a write*,
  - and at least *one access is non-atomic.*

	Var. Def.	Thread 0	Thread 1
✘	`int x;`	`… = x;        // read`	`x = 0xf0f0;  // write`
✘	`volatile int x;`	`… = x;    // volatile read`	`x = 0xf0f0;  // vol. write`
✘	`int x;`	`x = 42;       // write`	`x = 0xf0f0;  // write`
✔	`int x;`	`… = x + 42;  // read`	`… = x;       // read`
✔	`std::atomic<int> x;  // C++`	`… = x;      // atomic read`	`x = 0xf0f0; // atomic write`
✔	`_Atomic int x;       // C`	`… = x;      // atomic read`	`x = 0xf0f0; // atomic write`
✘	`std::atomic<int> x;`	`memcpy(buf, &x, sizeof(x));`	`x = 0xf0f0; // atomic write`

Undefined Behavior: Data Races

- Symptom of improperly synchronized concurrent execution.
  - Data Races defined as part of programming language Memory Consistency Model
- Often hardest to diagnose root cause:
  - Missing locking (quite common)
  - Calling the wrong function at the wrong time
  - Improperly synchronized lock-free algorithm
  - … anything goes.
- May result in anything from reading random data to program crashes.
  - Hard to understand impact: anything from “benign” to exploitable.
  - **Don’t take the risk:** make your programs data-race-free!

Dynamic Analysis: ThreadSanitizer

- **Programming Languages:** C, C++, (Rust)
- **False positives:** no
- **False negatives:** few + uncovered thread interleavings
- **Cost:** moderate

Description:

- *Compiler-instrumentation* based **data-race detector**.
- Part of GCC and LLVM Compiler Infrastructure.
- Usage: `clang -fsanitize=thread …`
- Usage: `gcc -fsanitize=thread …`
- See clang.llvm.org/docs/ThreadSanitizer.html for details.

```
$ pygmentize datarace.cpp
#include <thread>
int var;
void thread_func() {
  while (var++ < 100000);
}
int main(int argc, char *argv[]) {
  std::thread t1(thread_func);
  std::thread t2(thread_func);
  t1.join(); t2.join();
  return 0;
}
$ clang++ -O -fsanitize=thread datarace.cpp
$ ./a.out
==================
WARNING: ThreadSanitizer: data race (pid=921891)
  Read of size 4 at 0x55b8ea6096a8 by thread T2:
    #0 thread_func() <null> (a.out+0xd438b) (BuildId: a8cfff0e4f58e40cd9
    #1 std::thread::_State_impl<std::thread::_Invoker<std::tuple<void (*
    #2 <null> <null> (libstdc++.so.6+0xd44a2) (BuildId: c162fa2671dfc7cb

  Previous write of size 4 at 0x55b8ea6096a8 by thread T1:
    #0 thread_func() <null> (a.out+0xd43a9) (BuildId: a8cfff0e4f58e40cd9
    #1 std::thread::_State_impl<std::thread::_Invoker<std::tuple<void (*
    #2 <null> <null> (libstdc++.so.6+0xd44a2) (BuildId: c162fa2671dfc7cb

  Location is global 'var' of size 4 at 0x55b8ea6096a8 (a.out+0x14fb6a8)

  Thread T2 (tid=921894, running) created by main thread at:
    #0 pthread_create <null> (a.out+0x533fd) (BuildId: a8cfff0e4f58e40cd
    #1 std::thread::_M_start_thread(std::unique_ptr<std::thread::_State,
c162fa2671dfc7cb412fa26614863aa01ac3dae2)
    #2 __libc_start_call_main csu/../sysdeps/nptl/libc_start_call_main.h

  Thread T1 (tid=921893, finished) created by main thread at:
    #0 pthread_create <null> (a.out+0x533fd) (BuildId: a8cfff0e4f58e40cd
    #1 std::thread::_M_start_thread(std::unique_ptr<std::thread::_State,
c162fa2671dfc7cb412fa26614863aa01ac3dae2)
    #2 __libc_start_call_main csu/../sysdeps/nptl/libc_start_call_main.h
```

How to thwart security attacks?

Program hardening: "lightweight deterministic dynamic analysis", i.e. augment the original program with metadata (e.g. bounds of live objects or allowed memory regions) and insert access checks.

Software-based approaches:
compiler/runtime to transform applications to incorporate metadata management and access checking.

+ No hardware support required

- High-performance overheads

Hardware-based approaches:
HW support (registers/instructions) for metadata management and access checking.

+ "Low" performance overheads

- New hardware support required

Program Hardening: Software support

Adding instrumentation to release binaries deployed in production to become *fault-tolerant.*

- **Requirements:** acceptable performance properties, no false positives
- **Common trade-offs:** less useful error reports, more false negatives

Several compilers and standard libraries can build “hardened” binaries:

- *glibc _FORTIFY_SOURCE*
  - lightweight support for out-of-bounds accesses (crashes program on detection)
- *Stack Protection (-fstack-protector)*
  - *Stack canary detects “stack smashing” attempts (crashes program on detection)*
- *Automatic Stack Initialization (-ftrivial-auto-var-init)*
  - all stack memory will be automatically initialized (mitigation)
- Hardened Allocator (e.g. Scudo [llvm.org/docs/ScudoHardenedAllocator.html])
  - makes some heap memory-safety errors unexploitable (mitigation)
- **Growing set of hardening techniques to thwart attackers!**

Program Hardening: Memory Tagging

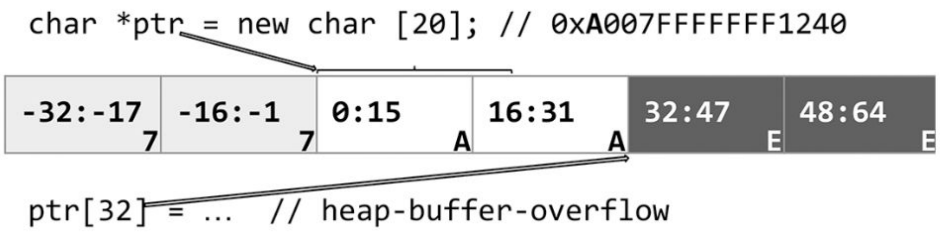
- Feature of recent Arm CPUs: “Memory Tagging Extensions (MTE)”
- Designed to *mitigate* memory-safety vulnerabilities of native code.
- Allows to assign “tags” to memory locations.
    - Tags are embedded into pointers.
    - If pointer tag does not match memory location’s tag → CPU raises exception.
- Can be used to implement low-cost memory-safety error detection.

📖 K. Serebryany, “ARM Memory Tagging Extension & How It Improves C/C++ Memory Safety,” USENIX 2019

Program Hardening: Memory Tagging

Detecting out-of-bounds accesses:

1. Allocator assigns random tag A to 32 bytes from `0x0007FFFFFFF1240`.
2. Returns pointer `0xA007FFFFFFF1240`.
3. Faulty access `ptr[32]` (off by 1) tries to access memory tagged with E (!=A) ⇒ CPU raises exception!

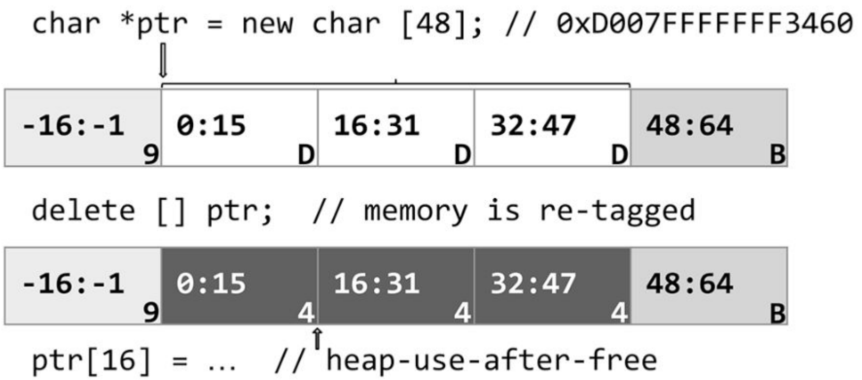


📖 K. Serebryany, "ARM Memory Tagging Extension & How It Improves C/C++ Memory Safety," USENIX 2019

Program Hardening: Memory Tagging

Detecting use-after-free accesses:

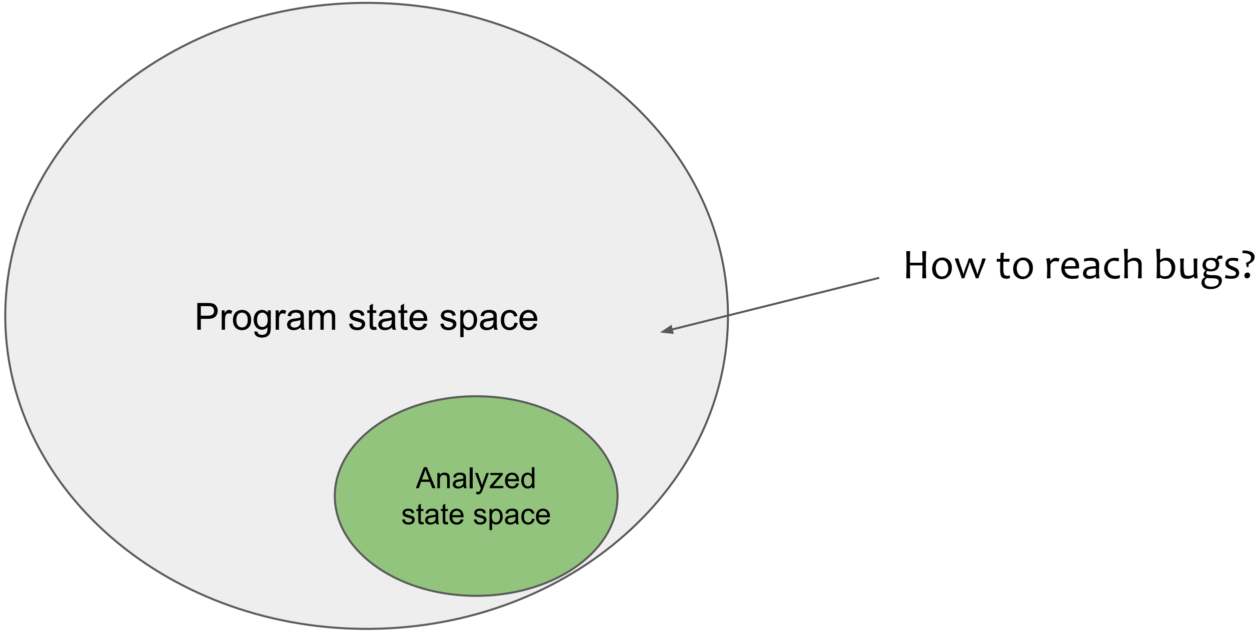
1. Allocator assigns random tag D to 48 bytes from `0x0007FFFFFFF3460`.
2. Returns pointer `0xD007FFFFFFF3460`.
3. Upon freeing memory, allocator re-tags memory with 4.
4. Faulty `ptr[16]` accesses memory with tag 4 (!= D) ⇒ CPU raises exception!



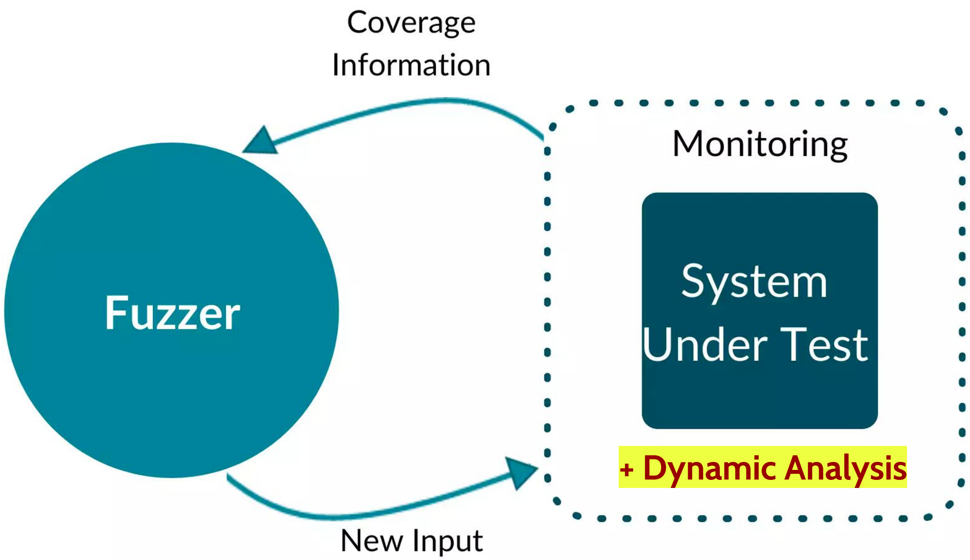
📖 K. Serebryany, “ARM Memory Tagging Extension & How It Improves C/C++ Memory Safety,” USENIX 2019

Dynamic Analysis: How to find lots of bugs fast?

- Only the state space that was covered during execution is analyzed.



Dynamic Analysis + Fuzzer ⇒ Find lots of bugs fast!



References

- UndefinedBehaviorSanitizer
- AddressSanitizer
- ThreadSanitizer
- MemorySanitizer
- MemoryTagging

Summary

- **Part I:** Faults and failures in software
  - Terminology and impact
- **Part II:** Program analysis trade-offs
  - Soundness, completeness, static vs. dynamic analysis
- **Part III:** Static analysis tools
  - Compiler warnings
  - Infer
  - SpotBugs
  - Clang Analyzer and Clang Tidy
- **Part IV:** Brief introduction to C
- **Part V:** Dynamic analysis tools
  - Undefined behavior
  - Dynamic binary instrumentation
  - Compiler-assisted instrumentation
  - Valgrind
  - Sanitizers
  - Program hardening

References

1. Ian Sommerville, “Software Engineering,” Global Edition, Pearson Education, Limited, 2015. URL: https://ebookcentral-proquest-com.eaccess.tum.edu/lib/munchentech/detail.action?docID=5831848
2. Heather Adkins, Betsy Beyer, Paul Blankinship, Piotr Lewandowski, Ana Oprea, and Adam Stubblefield, “Building Secure and Reliable Systems,” O’Reilly, 2020. URL: sre.google/static/pdf/building_secure_and_reliable_systems.pdf