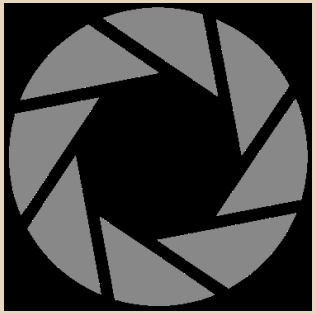


Advanced Fuzzing Topics

Offensive Computer Security
FSU CS Dept
Spring 2014

Outline

1. Taint analysis
2. Dynamic Taint Analysis
3. Code Paths
4. Symbolic representations
5. Advancements

“THE MASK” APT

attacked Kaspersky AV to privilege escalation & hide

- importance of finding vulns in own product

http://www.securelist.com/en/downloads/vlpdfs/unveilingthemask_v1.0.pdf

Crash Analysis / Taint Analysis

Once crashes are caused, determining if a vulnerability exists

Exploitability Analysis

Determine the *exploitability* of bugs.

- No accurate tools exist for vuln analysis.
  - !exploitable (http://msecdbg.codeplex.com/)
    - is a WinDBG extension
    - reports what is definately exploitable, probably exploitable, not exploitable, and unknown
    - *but is considered not very accurate*.
  - mona.py has an exploit generation

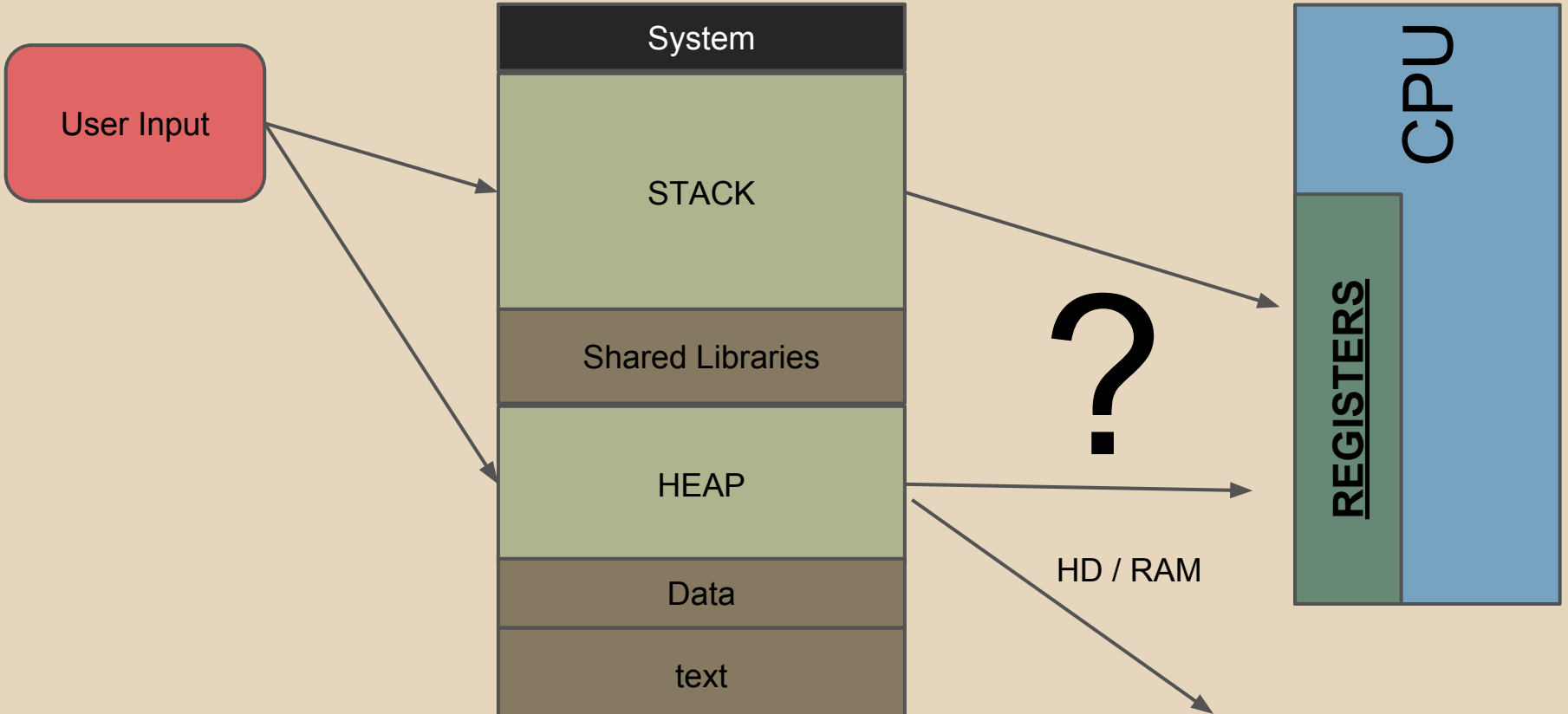
Taint Analysis

- Goal is to mark data originating from untrusted sources as *tainted*
- can be done statically / dynamically
- Two dependencies that determine taint:
  - Data flow dependencies
  - Control flow dependencies

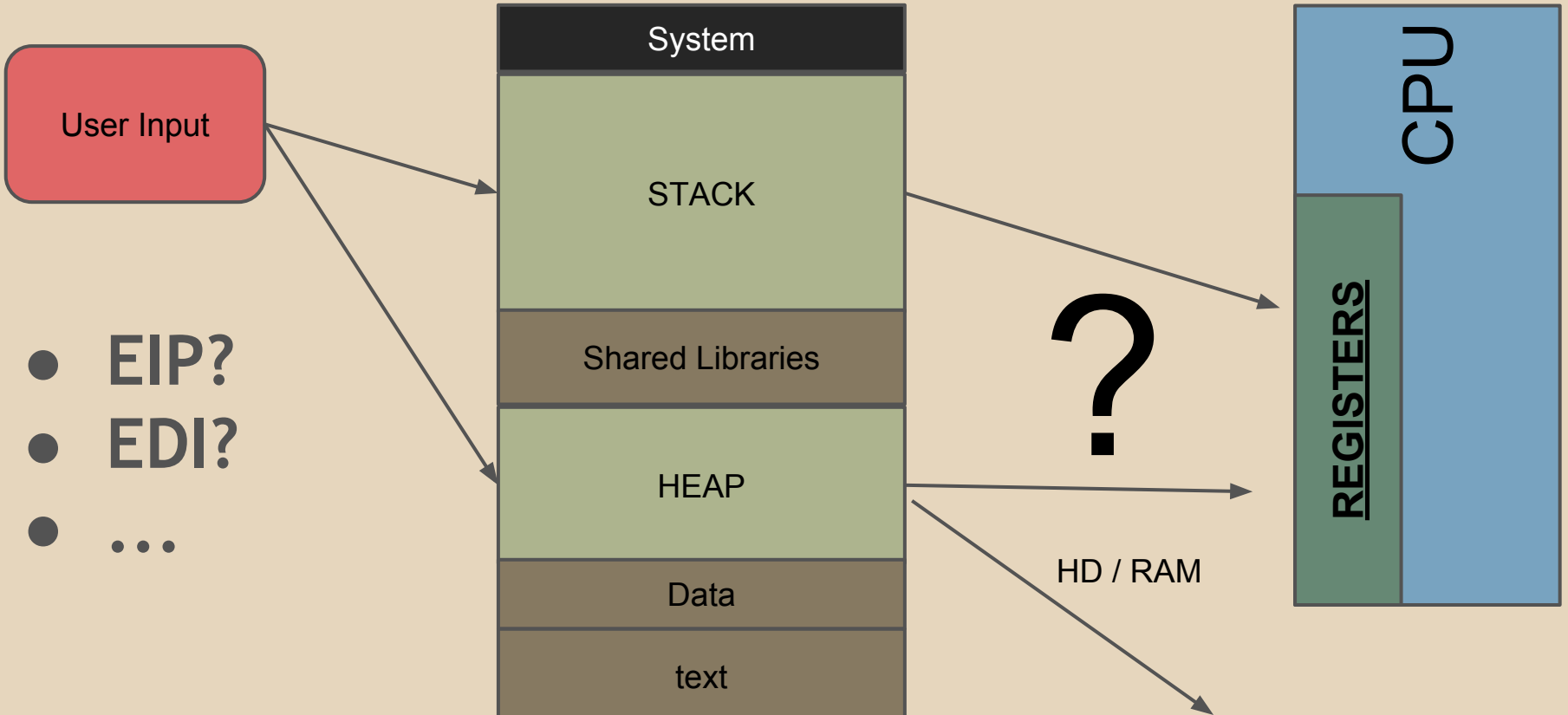
Taint Tracking Policies

- Just track data flow dependencies?
- Also data flow dependencies?
- Track taint after free() / garbage collection?
  - miss use after free or use uninitialized vulns.
- bytewise?
  - or even bitwise?

Taint Analysis



Taint Analysis



Taint Analysis

- Partial writes / Full writes
- operations like shifts, and, nand, ...
  - ones that destroy information

Stop tracking taint when

- variable overwritten by static /const value
- var assigned from untainted object

Simple Taint Analysis

User inputs are non-repeating patterns:
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5...

Simple Taint Analysis

Check if registers at crash contain any of these patterns(ENDIANNESS MATTERS HERE!)

EIP = 5Af6 (big endian) f65A (little endian)

RIP = Ag1Ag2Ag (big endian)
Agg21AAg (little endian (x86))

If confused, see (http://en.wikipedia.org/wiki/Endianness)

Problem

- Patterns won’t work in all applications
- method fails on any transforms of user input
- encoding / decoding / expansion /etc…

This is a dumb approach, but effective.

Data flow dependencies

```
//x is tainted
 y = 2;
 z = x + y;
//z is tainted
```

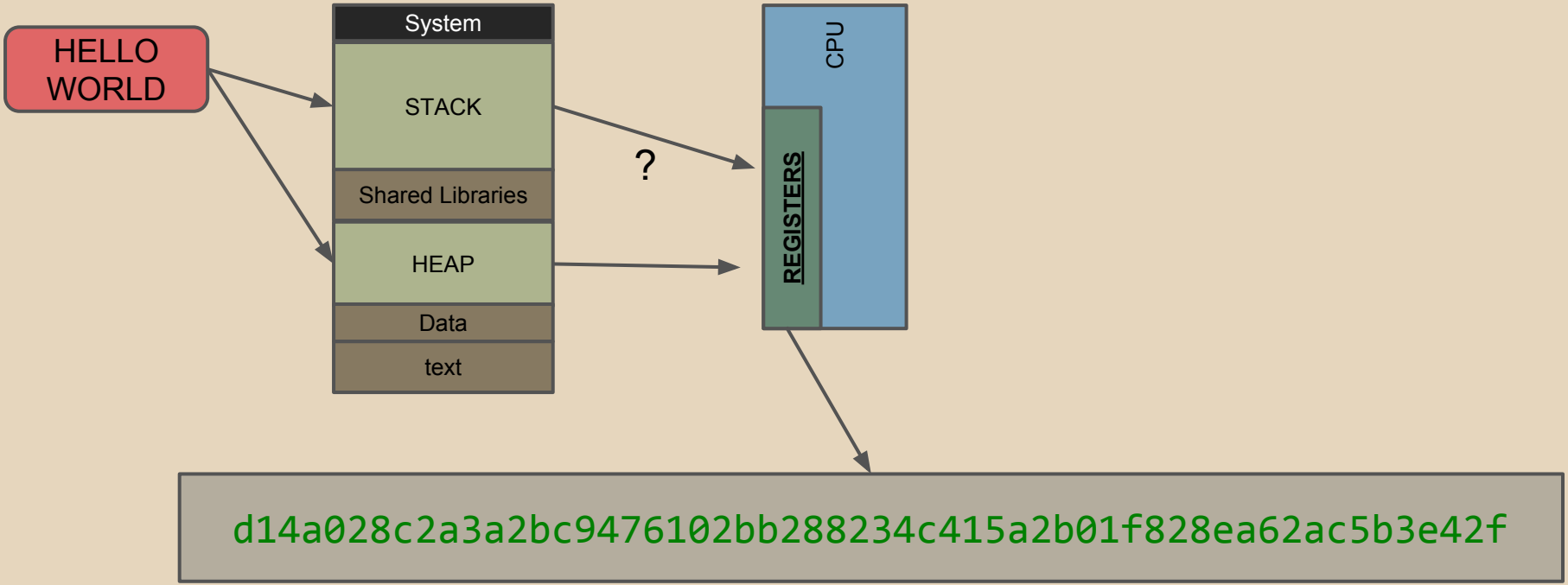
From: http://diyhpl.us/~bryan/papers2/paperbot/e15bb28f0dc692c053f64bb48b879ab3.pdf

Control flow dependencies

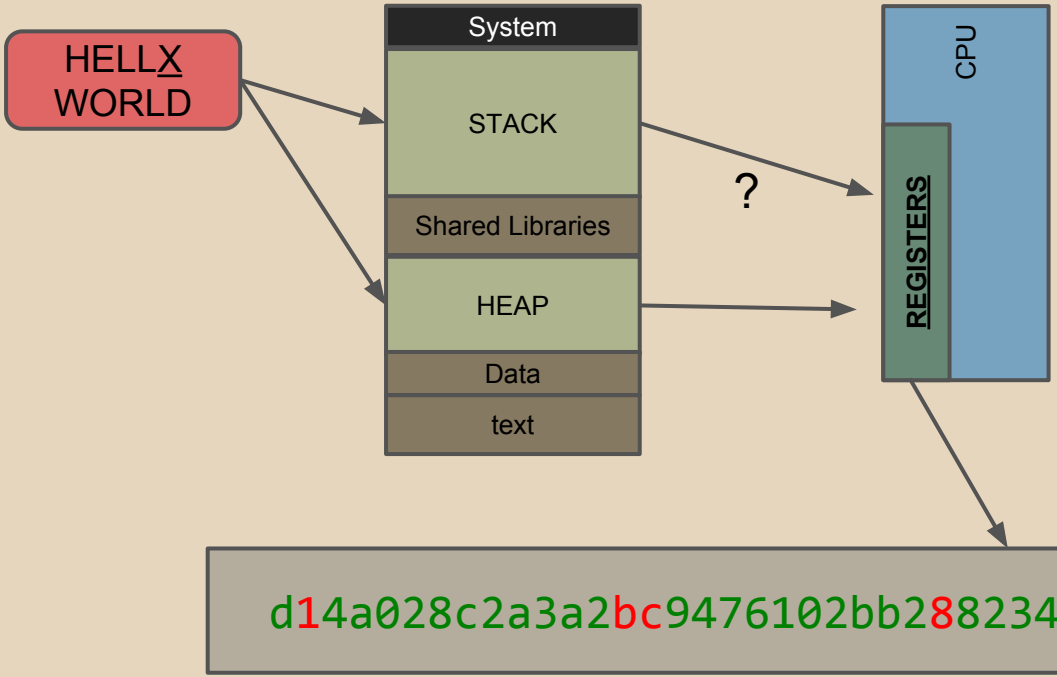
```
//x is tainted
if (x > 1) y = 1 else y =2;
//y is tainted b/c influenced by x
```

From: http://diyhpl.us/~bryan/papers2/paperbot/e15bb28f0dc692c053f64bb48b879ab3.pdf

Taint Analysis



Taint Analysis (For Differential Cryptanalysis)



Throwing patterns here doesn't work.

How would we pull this off?

Dynamic Taint Analysis (DTA)

DTA steps

1) Tag data originating from an unsafe source as *tainted*

2) track *tainted* data during execution

3) identify and prevent unsafe usage of *tainted* data.

REMEMBER THESE

DTA

Usually requires fine-grained control of the system

- QEMU (emulator)
- Debuggers
- etc…

DTA Uses

- Bug hunting
- Detecting 0-days
  - **Argos** http://www.few.vu.nl/argos/?page=1

Code Paths

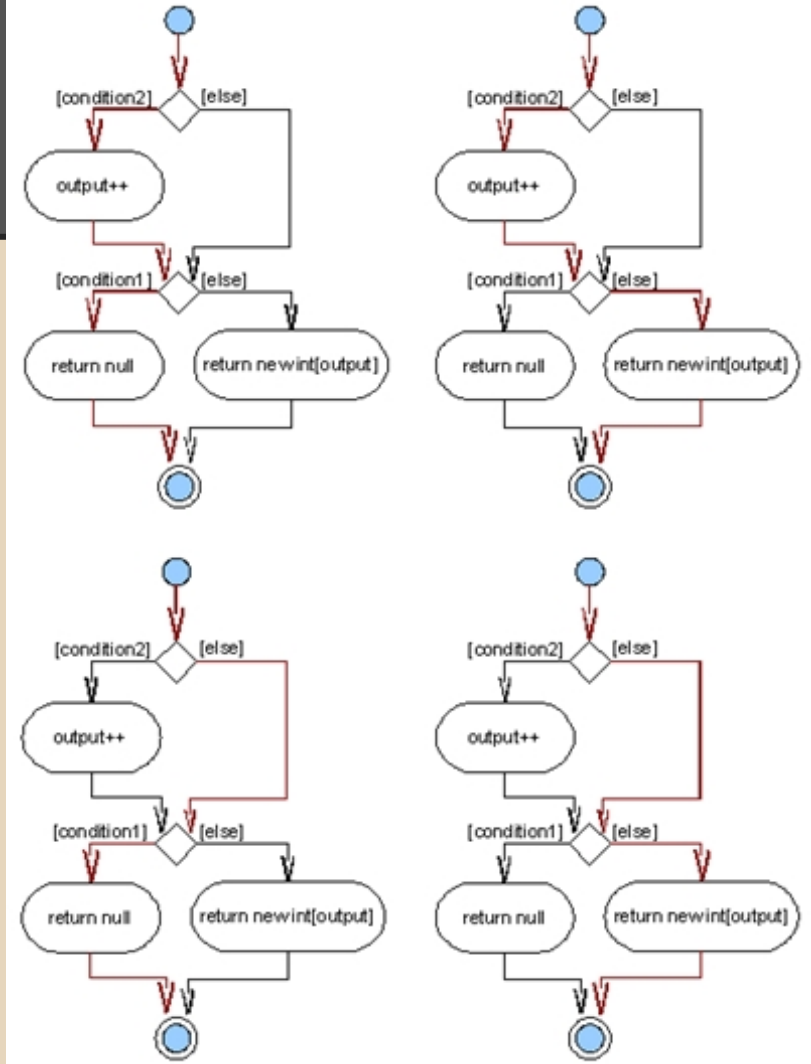
and avoiding explosions

Dynamic Analysis Problems

1. Code Coverage
2. Symbolic representation & Path Explosion
3. Differential fuzzing

Code Coverage

- Cover interesting Paths
- Branches are expensive
  - imposes exponential cost
    - avoiding loops
    - avoiding already covered paths
    - required careful optimization
- ++Reasoning about code



Code paths and reality

90% of execution time is spent in loops

- Tons of optimization research done here
- This alone is the subject of many PhD dissertations
  - branch prediction

A vulnerable code path might be traveled < 0.1% of the time on average.

Symbolic Representation / Constraints

```
if (x > 0 && x < 1600)

    // does testing 1,2,3,4,..1598,1599.. matter?
```

Basic Symbolic Representation

Variables are represented by symbols

- allow symbols to take any value

replace concrete operations with operations that manipulate symbolic values

- special focus on branches
  - branches go one of two ways

Basic Symbolic Representation

When program branches on a symbolic value:

- system follows both branches at once
  - keeps track of "***path constraint***" *that determines the branch behavior*

When a code path ends / bug is found:

- *test case can be quickly generated by solving all of the path constraints!*
  - *constraint solvers*

Basic Symbolic Representation

Problems:

-Scales poorly

-high number of branches = explosion of code paths

Basic solutions to these ^

+parallel

+heuristics

Intermediate Representations (IR/IL)

- A frontend (language) that facilitates the RE/analysis of code
  - machine code
  - source code
  - malware
- The IR/IL can be:
  - pseudo code
  - or other executable code *(Forward Symbolic Execution)*

SimpIL (Simple Intermediate Language)

General language for symbolic execution / taint analysis

http://users.ece.cmu.edu/~ejschwar/papers/oakland10.pdf

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).

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.

How SimpIL works

1. Program = sequence of numbered statements
2. statements = assignments, assertions, jumps, and conditional jumps

```
if (x <1)
        x = 0;
if (blah)
        foo();
return();
```

How SimpIL works

3. Expressions are things that have no side-effects (i.e. do not change the program state).

How SimpIL works

$\Diamond_b$ = These are binary operators:

- +, -, /, *, ^, &(bitwise and), …
- *Not to be confused with bitwise operators*

$\Diamond_u$ = These are unary operators:

- !, ++, --, &(address of), *(indirection), ~ (one's compliment), type-casts, sizeof()
- See: http://en.wikipedia.org/wiki/Unary_operation#C_family_of_languages

SimpIL (Simple Intermediate Language)

General language for symbolic execution / taint analysis

http://users.ece.cmu.edu/~ejschwar/papers/oakland10.pdf

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).

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.

SimpIL (Simple Intermediate Language)

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

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

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

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

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

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

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

Figure 1: Operational semantics of SIMPIL.

Lets see an example

Following (http://users.ece.cmu.edu/~ejschwar/papers/oakland10.pdf)

- Assume 32 bit int values
- assume everything is well-typed (type checking is omitted)

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

Simple right?

Example 1. *Consider evaluating the following program:*

```
1  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.*

(This won't be on the exam)

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

Figure 3: Evaluation of the program in Listing 1.

Symbolic Execution

+ code coverage
+ concolic techniques

- heavyweight
- complicated!
- not always worth it

Recent publications / State of the Art

“Adaptive Random Testing” (~2004)

http://www.utdallas.edu/~ewong/SYSM-6310/03-Lecture/02-ART-paper-01.pdf

- distributes test cases more evenly in test space
  - outperforms ordinary random testing
- Widely cited paper
  - great discussion of it here http://blog.regehr.org/archives/1039

“Adaptive Random Testing” (~2004)

http://www.utdallas.edu/~ewong/SYSM-6310/03-Lecture/02-ART-paper-01.pdf

- Widely cited paper
  - great discussion of it here http://blog.regehr.org/archives/1039
- Talks about quality of constraints, and how to distribute test cases.

“Smart” Fuzzing

“A Taint Based Approach for Smart Fuzzing”

(VUPEN & Grenoble University)

- http://diyhpl.us/~bryan/papers2/paperbot/e15bb28f0dc692c053f64bb48b879ab3.pdf
- use taint analysis to judge code coverage, and fuzzing efficiency
  - differentially adjust fuzzing to improve coverage
    - depending on how many path constraints have been exhausted

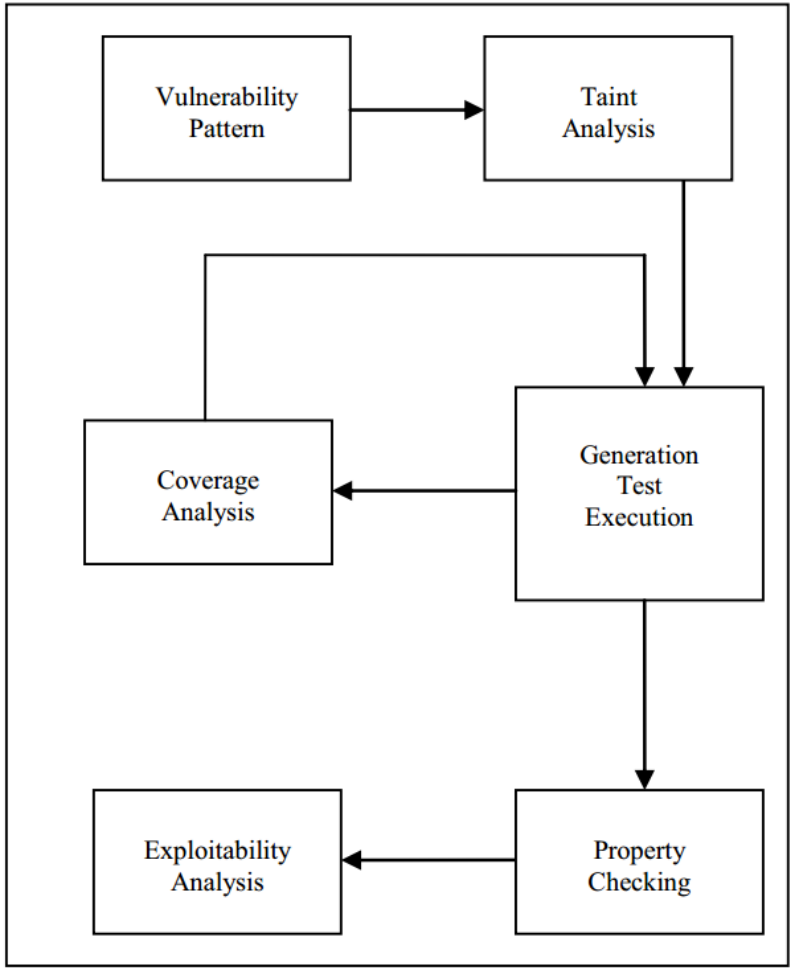
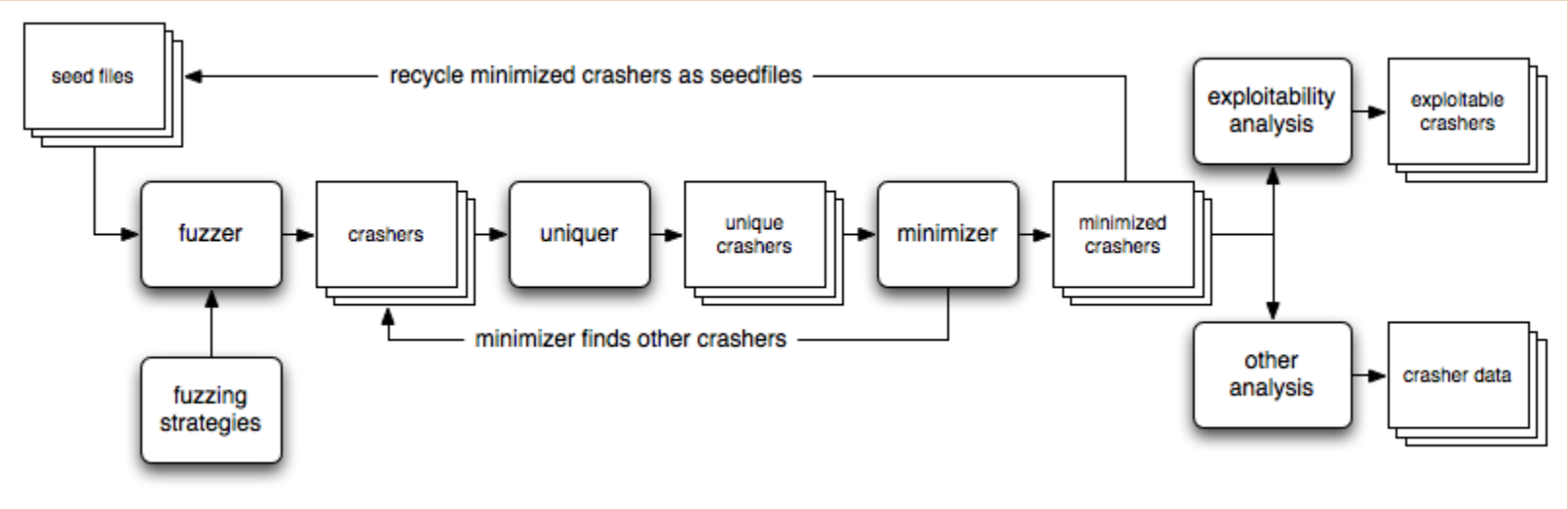


Figure 1. Proposed approach.

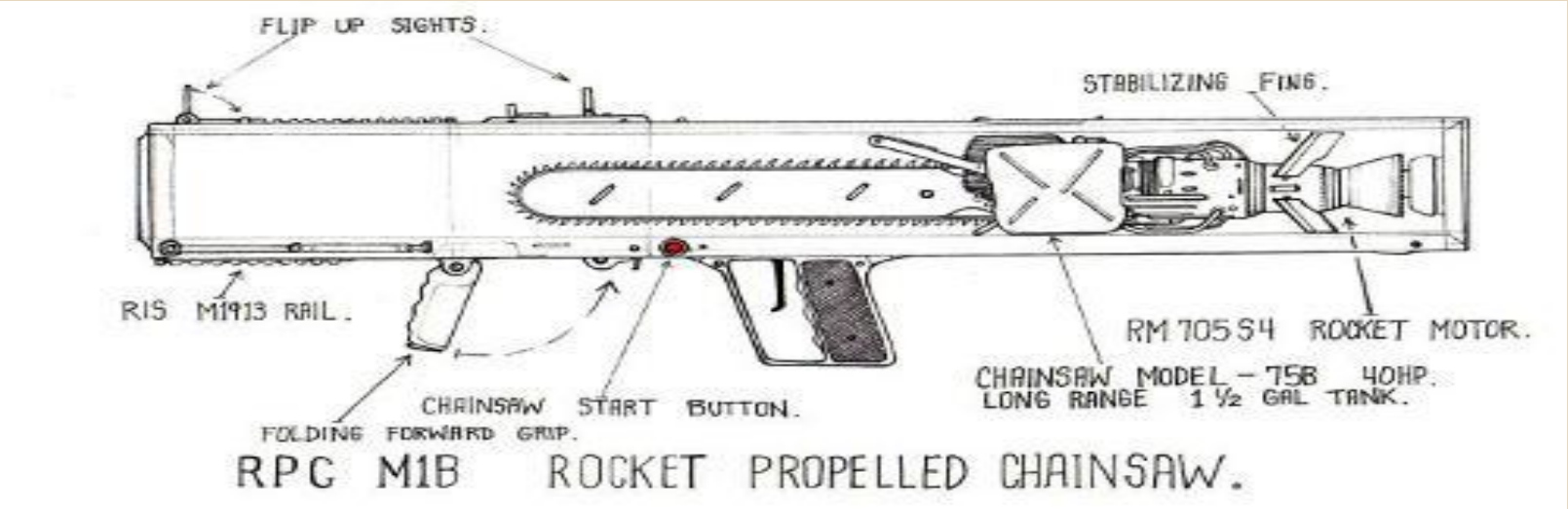
Advancements / State of Art

“Rocket Propelled Chainsaw”



Advancements / State of Art

“Rocket Propelled Chainsaw”



Advancements / State of Art

“Unleashing MAYHEM on Binary Code”

http://www.cse.psu.edu/~tjaeger/cse597-s13/docs/binary_mayhem_oakland_12.pdf

- found 1200 bugs (http://lwn.net/Articles/557055/)
  - 29 published 0-days
- novel hybrid symbolic execution

(other) Advancements / State of Art

- given target code chunk, “bubbles” back up the code paths to determine if reachable and how.
  - can’t remember who / what :(

GDSL Toolkit: Advancement in IRs

http://www2.in.tum.de/bib/files/simon14gdsl.pdf

- Great discussion on IR existing work and pitfalls.
  - most only cover a subset of instruction set (e.g. x86)

DISCLAIMER on State of Art

This by no means is a comprehensive coverage of the “state of the art” of fuzzing, DTA, forward symbolic execution, etc...

Questions?

Read: Differential Testing for Software (http://www.cs.dartmouth.edu/~mckeeman/references/DifferentialTestingForSoftware.pdf)

Read: Attaching the Rocket to the Chainsaw https://www.cert.org/blogs/certcc/2013/09/putting_the_rocket_on_the_chai.html

BFF

Basic Fuzzing Framework (Linux / Mac)

http://www.cert.org/vulnerability-analysis/tools/bff.cfm

FOE

Failure Observation Engine (Windows fuzzing)

http://www.cert.org/vulnerability-analysis/tools/foe.cfm