Program Analysis

Lecture 11: *SimpIL and Dynamic Taint Analysis*
Winter term 2011/2012

Prof. Thorsten Holz

Announcements

- Today only 45 minutes lecture
  - I have a meeting at 10am...
- Solution for Christmas challenge and last exercise
  - Two solution attempts were handed in
  - Missing solution for last exercise

Outline

- *SimpIL*
  - Definitions and examples
- Dynamic taint analysis (*next lecture*)
  - Intuition
  - Definitions and examples

SimpIL

Motivation

- We now focus on two import techniques
  - Taint analysis
  - Symbolic execution
- To understand these concepts we do not use x86 assembler, but an *intermediate language* (*IL*)
  - Avoids complexity
  - Enables us to reason about code

SimpIL

- **Simp**le **I**ntermediate **L**anguage
  - Useful to learn concepts
  - Presented in a paper by Schwartz et al. ("All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)" - IEEE S&P'10)
- Today's lecture is mainly based on this paper, link is available in Moodle
- REIL is an intermediate language developed by zynamics

SimpIL

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* $\lozenge_b$ *exp* | $\lozenge_u$ *exp*
		| *var* | get_input(src) | v
$\lozenge_b$	::=	typical binary operators
$\lozenge_u$	::=	typical unary operators
value v	::=	32-bit unsigned integer

Language definition

SimpIL

- Expressions in SimpIL are side-effect free (i.e., they do not change the program state)
  - x86 has many side effects that complicate analysis
- Only expressions (constants, variables, etc.) that evaluate to 32-bit integer values
- No type-checking semantics in the language, we assume things are well-typed (i.e., operands have the correct type)

SimpIL Examples


```
1: x := 2 * get_input(·)
```

SimpIL Examples


```
1: x := 2 * get_input(·)
```

```
1: x := 2
2: y := 5 + x
```

SimpIL Examples


```
1: x := 2 * get_input(·)
```

```
1: x := 2
2: y := 5 + x
```

```
1: a := get_input(·)
2: if a > 42 then goto 3 else goto 5
3:   a := a - 5
4:   b := a
5: c := 7
6: assert(a < 23)
```

Operational Semantics

- Next we need to define how to operate on SimpIL, we need to define how a program is actually executed
- This is done using specific operational semantics
  - We need to first define the execution context, i.e., some kind of abstract execution framework
  - And then we can define how to operate on this framework
- This looks complex at first, but is necessary to be precise and enables a comprehensive analysis

Definitions I

- Execution context is defined by five parameters
  - List of program statements **Σ**
  - Current memory state **μ**
  - Current value for variables **Δ**
  - Program counter **pc**
  - Current statement **ι**

Definitions II

- Σ, μ, and Δ contexts are maps
  - Σ[y] denotes the program statement y
  - μ[z] maps to value at memory address z
  - Δ[x] denotes the current value of variable x
- Updating a context variable **x** with value **v** is denoted as **x** ← **v**
  - Δ[x ← 10] denotes setting the value of variable x to the value 10 in context Δ

Definitions III

- $\boldsymbol{\mu}, \boldsymbol{\Delta} \vdash \mathbf{e} \Downarrow \mathbf{v}$ denotes evaluating an expression **e** to a value **v** in the current state given by memory state **μ** and variables **Δ**
- Based on these definitions, we can now define the actual operational semantics

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

- Rules are read bottom to top, left to right

Operational Semantics I

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

$\mu,\Delta \vdash e \Downarrow v$ denotes evaluating an expression e to a value v in the current state given by memory state μ and variables Δ

Operational Semantics I

$$\frac{\mu, \Delta \vdash e \Downarrow v \quad \Delta' = \Delta[var \leftarrow v] \quad \iota = \Sigma[pc+1]}{\Sigma, \mu, \Delta, pc, var := e \leadsto \Sigma, \mu, \Delta', pc+1, \iota} \text{ ASSIGN}$$

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

$\mu,\Delta \vdash e \Downarrow v$ denotes evaluating an expression e to a value v in the current state given by memory state μ and variables Δ

Operational Semantics II

Systems Security
Ruhr-University Bochum

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

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

Operational Semantics II

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

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

$$\frac{\mu, \Delta \vdash e \Downarrow v_1 \quad \iota = \Sigma[v_1]}{\Sigma, \mu, \Delta, pc, \text{goto } e \rightsquigarrow \Sigma, \mu, \Delta, v_1, \iota} \textsc{Goto}$$

Operational Semantics III

$$\frac{\mu, \Delta \vdash e \Downarrow v \quad v' = \lozenge_u v}{\mu, \Delta \vdash \lozenge_u e \Downarrow v'} \ \textsc{Unop}$$

$$\frac{\mu, \Delta \vdash e_1 \Downarrow v_1 \quad \mu, \Delta \vdash e_2 \Downarrow v_2 \quad v' = v_1 \lozenge_b v_2}{\mu, \Delta \vdash e_1 \lozenge_b e_2 \Downarrow v'} \ \textsc{Binop}$$

Operational Semantics III

Systems Security
Ruhr-University Bochum

$$\frac{\mu, \Delta \vdash e \Downarrow v \quad v' = \Diamond_u v}{\mu, \Delta \vdash \Diamond_u e \Downarrow v'} \textsc{Unop}$$

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

$$\frac{\mu, \Delta \vdash e \Downarrow v_1 \quad v = \mu[v_1]}{\mu, \Delta \vdash \text{load } e \Downarrow v} \textsc{Load}$$

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

Operational Semantics IV

$$\frac{v \text{ is input from } src}{\mu, \Delta \vdash \text{get\_input}(src) \Downarrow v} \textsc{Input}$$

Operational Semantics IV

$$\frac{v \text{ is input from } src}{\mu, \Delta \vdash \text{get\_input}(src) \Downarrow v} \textsc{Input}$$

$$\frac{}{\mu, \Delta \vdash var \Downarrow \Delta[var]} \textsc{Var}$$

$$\frac{}{\mu, \Delta \vdash v \Downarrow v} \textsc{Const}$$

Operational Semantics IV

$$\frac{v \text{ is input from } src}{\mu, \Delta \vdash \text{get\_input}(src) \Downarrow v} \textsc{Input}$$

$$\frac{}{\mu, \Delta \vdash \mathit{var} \Downarrow \Delta[\mathit{var}]} \textsc{Var}$$

$$\frac{}{\mu, \Delta \vdash v \Downarrow v} \textsc{Const}$$

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

Rule Evaluation

Rules are quite complex at first, evaluating rules is also complex:

```
1: x := 2 * get_input(◦)
```

is evaluated for the input *20* to:

Rule Evaluation

Rules are quite complex at first, evaluating rules is also complex:

```
1: x := 2 * get_input(·)
```

is evaluated for the input *20* to:

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

Since the ASSIGN rule requires the expression e in var := e to be evaluated, we have to recurse to other rules (BINOP, INPUT, CONST) to evaluate the complete expression

Discussion

- SimpIL looks complex at first glance
  - ... also at second glance
- But we need to be precise to *reason* about programs
  - We need to define operational semantics such that we can actually work with the language
  - Transitions need to be clear and unambiguous

Tools

- BAP (http://bap.ece.cmu.edu/) and BitBlaze (http://bitblaze.cs.berkeley.edu/) implement a variant of SimpIL to perform analysis
  - Source code is available, feel free to play with it
- REIL by zynamics is very similar and there is also an analysis framework to work on top of REIL
- Can be a topic for a master's thesis ☺

Questions?

Contact:
Prof. Thorsten Holz
thorsten.holz@rub.de
@thorstenholz on Twitter

More information:
http://syssec.rub.de
http://moodle.rub.de

Sources

- Paper by Schwartz et al.: "All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)" - IEEE S&P'10
  - http://www.ece.cmu.edu/~ejschwar/papers/oakland10.pdf