TECHNISCHE
UNIVERSITÄT
DRESDEN

Faculty of Computer Science Institute for System Architecture, Operating Systems Group

Security - Verification

Benjamin Engel

Dresden, 2011-01-11

Overview

- Type systems
- Information flow
- System software verification

Recap

- Bell – La Padula model
  - Subjects and objects have a security level (confidential, top secret, ...) associated
  - Security levels are ordered

- Access to an object is granted if the accessor's label dominates the accessed object

- Too limited for many real-world scenarios (model not expressive enough)

→ Type Systems

Type Systems – Introduction

- Suitable to reason on programs and to prove properties like non-interference
- Most prominent example: **Data type systems** in programming languages (C, Java)
  - Define meaning of data (bit patterns) and which operations are allowed
  - Assure data type compatibility
  - Normally this should not type check (without automatic type conversion):

```
int i = 0;
float f = i;
```

Theory : Data Type Systems

- If e_1 is an expression of type integer and e_2 also, then $e_1 + e_2$ can also be typed as integer
- Formal inference rule:

$$\frac{\gamma \vdash e_1 : \text{int} \qquad \gamma \vdash e_2 : \text{int}}{\gamma \vdash e_1 + e_2 : \text{int}}$$

γ – type environment

$e_1, e_2, e_1 + e_2$ – expressions

- Rules for statements, expressions, variables, ...
- A program is said to be well-typed if it follows the typing rules

Type Inference vs. Type Checking

- Some expressions are initially typed
- Types are inferred by typing rules

$\gamma = \{11, 12 : \texttt{int}\}$

```
let a = 11 in
let b = 12 in
let c = a + b
```

- All types are known in advance
- Soundness proof of the types through typing rules

$\gamma = \{11, 12, a, b, c : \texttt{int}\}$

```
int a = 11;
int b = 12;
int c = a + b;
```

Flow Insensitivity vs. Flow Sensitivity

- Statically typed (type of expressions never changes)
- Order of evaluation does not matter

- Dynamically typed (types might change)
- Evaluation order can influence γ, (type environment)

$$\frac{\gamma \vdash e_1 : \text{int} \qquad \gamma \vdash e_2 : \text{int}}{\gamma \vdash e_1 + e_2 : \text{int}}$$

$$\frac{\gamma \vdash e_1 : \text{int}, \gamma' \qquad \gamma' \vdash e_2 : \text{int}, \gamma''}{\gamma \vdash e_1 + e_2 : \text{int}, \gamma''}$$

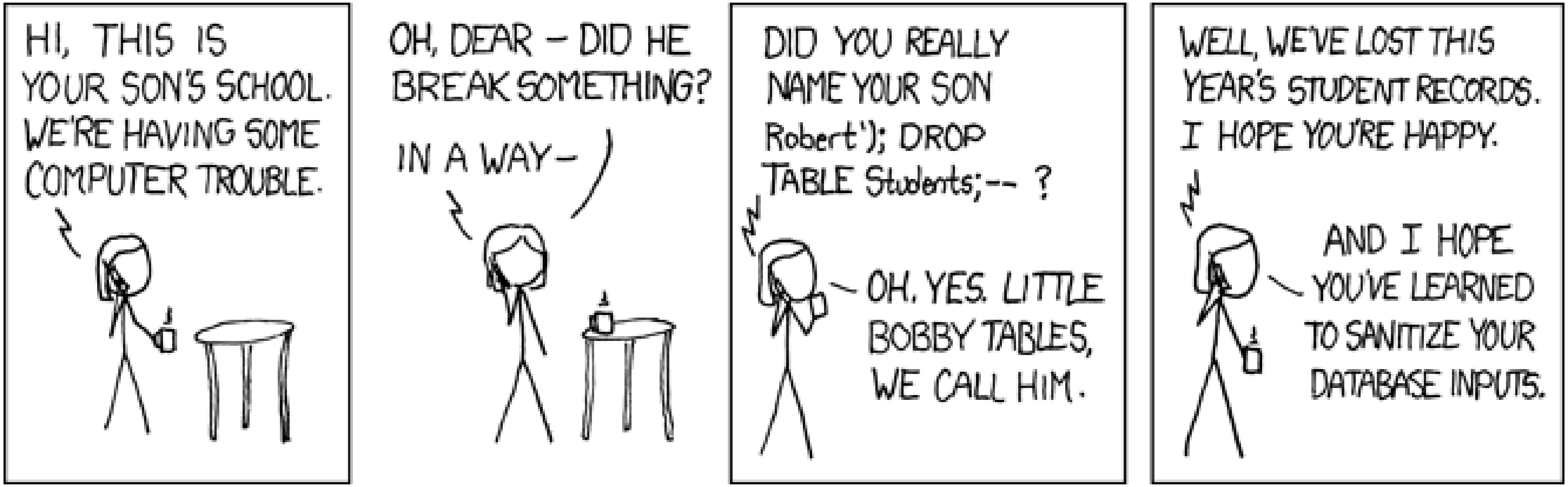
Other Type Systems

- Const inference
  - Compilers can optimize more heavily if constness (read only) of a variable is known
  - Programmer can declare a variable const, but this is cumbersome and error-prone
  - automatically infer as many “consts” as possible using a feasible type system

- non-NULL inference
  - Compile-time detection if null pointer dereferences
  - Proving/inferring a pointer to be non-NULL at least removes the check

Bobby Tables

- User-supplied input is not trustworthy
- Should not been used unless sanitized



http://xkcd.com/327

Format String Vulnerabilities

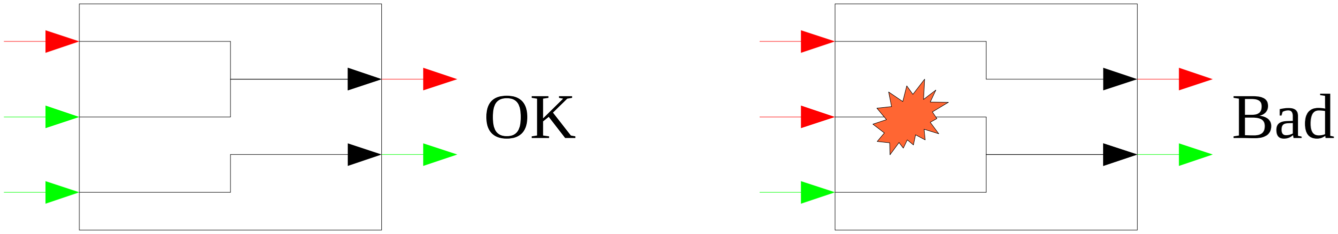
- Type system for **taint analysis**
  - User supplied input data is typed tainted
  - Might get declassified as untainted by sanitize functions
  - If tainted data is used as a format string
    → Type error or bug

good: `printf ("%s", buffer);`

bad: `printf (buffer);`

Non-interference

- Programs as input-output model
- Data is classified as confidential or public
- Confidential data is typed as *high (H)*, public data as *low (L)*
- Inputs and Outputs are typed *H* or *L*
- *Low*-typed outputs must not depend on *high*-typed inputs
- Observing *low*-typed outputs reveals no information about *high*-typed inputs



Confidentiality Through Type Systems

- Variables and expressions are typed according to the information they contain
- Security levels form a lattice (partially ordered set), e.g. {*low*, *high*} with *low* ≤ *high*
- Confidentiality is preserved if no *high* classified input variable writes to a *low* output
- e.g. e_1 : low and e_2 : high → $e_1 + e_2$: high

$$\frac{\begin{array}{l}\gamma \vdash e_1 : l_1 \\ \gamma \vdash e_2 : l_2\end{array}}{\gamma \vdash e_1 + e_2 : l_1 \cup l_2}$$

l_1, l_2 – security level

$\cup$ – least upper bound

Example: Well-typed and Untypeable

medium := low;	low := high;
high := medium;	low := 0;

- w.r.t. confidentiality left example well-typed
  - Assignments: security level increases, thus no information leakage
- Right example: not typeable, first assignment already break it
  - Never copy data from a variable with a higher security level to one with a lower level
  - Code is still secure (no information is leaked - why?), but this cannot be **proven** by the type system

TECHNISCHE
UNIVERSITÄT
DRESDEN

Static Type Systems (flow-insensitive)

- Secure programs with **temporal** information leakage cannot be typed (in general) with a static type system

`low = high;`	to type this statement, low would have to be typed high
`low = 0;`	this fixes the temporal leakage, but static typing cannot do that

- Static single assignment form (SSA) helps as long as no pointers are involved (aliasing)

```
low_1 = high;
low_2 = 0;
```

Dynamic Type Systems (flow-sensitive)

- Types (security level) of variables and expressions change over time

  lattice: $\perp < L < H < \top$

  `low = high;` γ = {low:H; high:H}

  `low = 0;` γ = {low:⊥; high:H}

- \+ more programs are typeable
- – flow-sensitive → loops and function calls are now a **real** problem

- Secure, but not typeable (semantic gap):

```
low = high;
low -= high;
```

Flow-sensitive Type System

- Recap:
  - Bell La-Padula: security levels, linearly ordered
  - Static type systems
    - finite lattice
    - Static typing of variables and expressions
    - No halting problem (loops)
    - Flow-insensitive → cannot cover temporal information leakage
  - Dynamic type systems
    - Type of variables and expressions might change → solves temporal leakage
    - Halting problem now an issue
    - Very closely related to data flow analysis

Types and Labels: Real Life Example

- HiStar OS: Explicit information flow
  - Small kernel (18.000 SLOC)
  - Designed towards information flow security
- Loki: Tagged memory
  - every memory word has a tag field associated
  - Fine-grained access control on physical memory
  - FPGA prototype, checks tags in CPU pipeline
- LoStar: HiStar + Loki
  - Monitor beneath kernel, translates HiStar labels to Loki tags, kernel no longer trusted

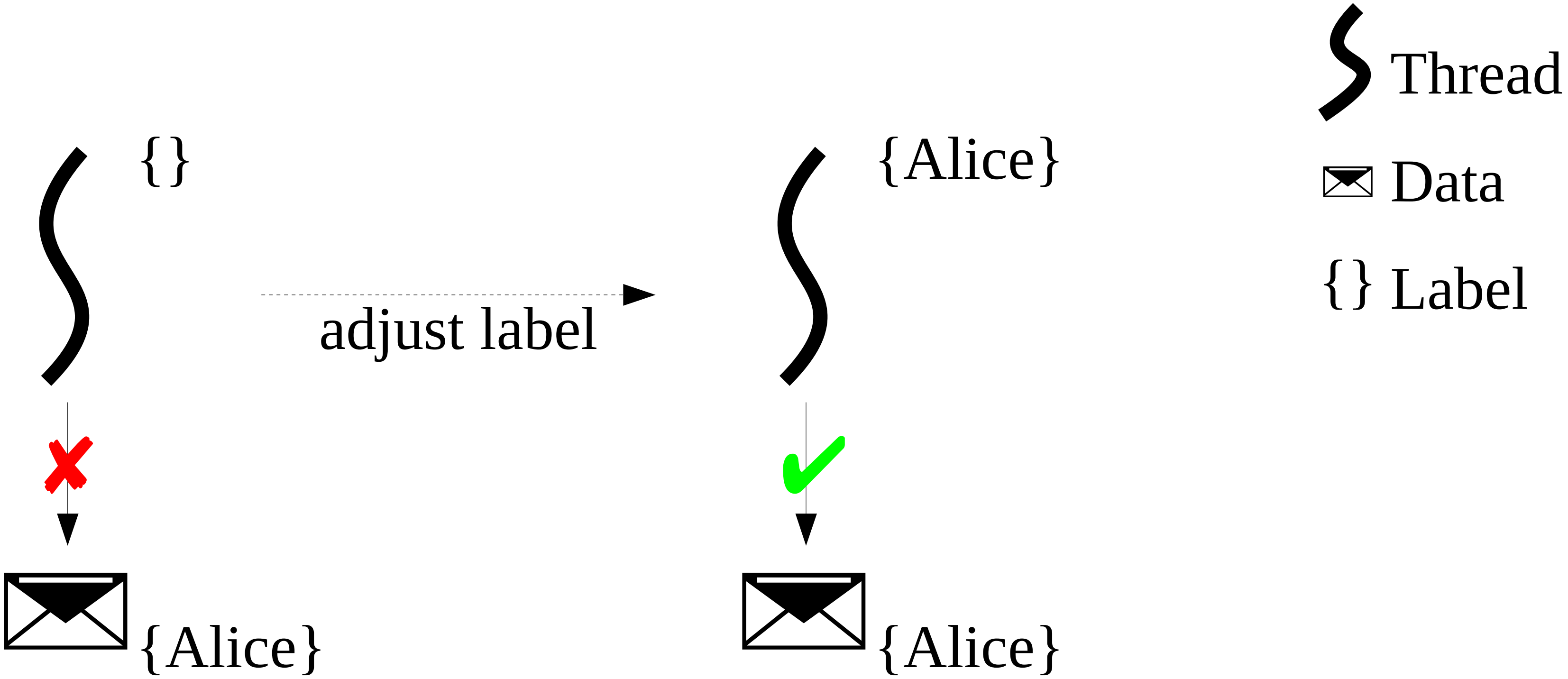
HiStar: Introduction

- Strict information flow control
- Few kernel objects: segments, address spaces, devices, threads, containers, gates
- Most UNIX functionality is implemented in a user-level library
- Labels: Set of categories
  - Attached to kernel objects
  - Describe security policy (read/write access)
- Categories: Describe kind of data (meaning)
  - Processes, threads, UNIX file descriptors, UIDs

HiStar: Labels

- Thread's label: which data it might access
- Threads can add categories to their label to access secret data
- They cannot remove them later → security
- Threads have a clearance (set of categories), limiting allowed accesses
- System calls on kernel objects:
  - Kernel knows in advance which information flow might occur
  - Use labels of effected objects to determine if this operation is allowed or not

Example of HiStar Labels



{}

adjust label

{Alice}

✗

✓

{Alice}

{Alice}

Thread

✉ Data

{} Label

Access denied,
label not sufficient

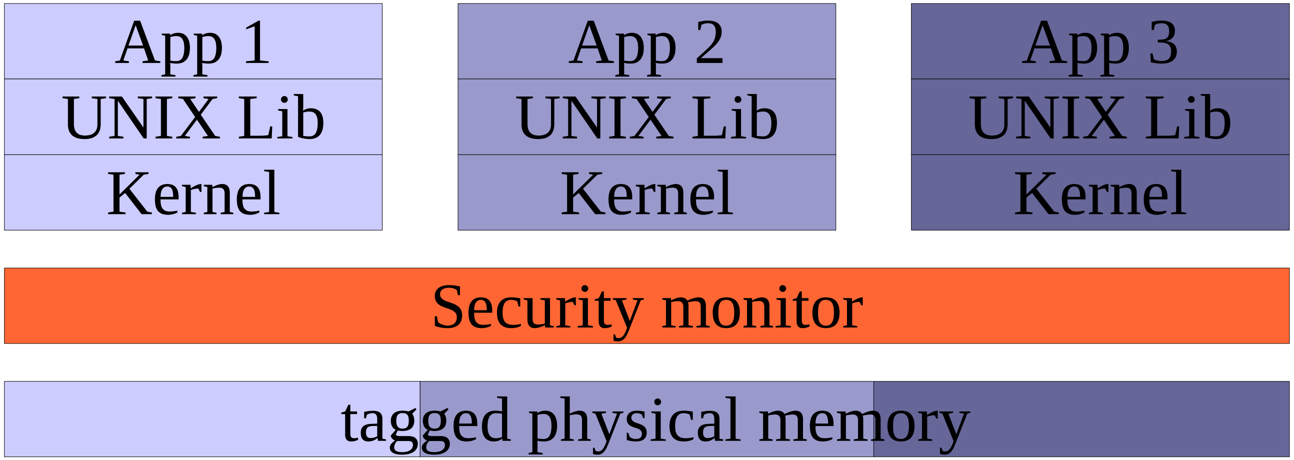
Access granted

Loki: Tagged Memory

- Move labels from software into hardware
- Modified SPARC processor, 7 stage pipeline
- In-CPU permission(tag) cache, accessed at instruction fetch and loads/stores
- Use tagged memory (32 bit word + 32 bit tag) → 100% memory overhead
- Multi-granular tagging scheme (per page, per word) for fine-grained access control
- Special monitor mode to modify memory tags/permission cache

HiStar + Loki = LoStar

- Thin security monitor is put beneath the kernel, translates labels to tags
- One <u>logical</u> kernel per thread
- Benefit: even a compromised kernel cannot afflict unrelated processes



App 1 | App 2 | App 3

UNIX Lib | UNIX Lib | UNIX Lib

Kernel | Kernel | Kernel

Security monitor

tagged physical memory

Microkernels and Data Flow Analysis

- Microkernel-based Operating Systems:
  - Already well-defined components at user level
  - Strong isolation, thin interface (if possible)

- Confidentiality and integrity concerns are expressible in terms of information flow
  - Private data should never “flow” (be revealed) to an unauthorized subject
  - No (unauthorized) data should “flow” (be written) to objects with higher integrity (e.g. system files)

- Proving **Non-interference** of components

Where does data come from …
… and where does it go

From Type Systems to Data Flow

- Generalize lattice → universal lattice (L,∪,∩)
  - Every variable gets an unique identifier
  - L = power set of the set of all IDs
  - 2 variables a and b → L = {∅, {a}, {b}, {a,b}}
- Security level (or label, former *low* or *high*) of an expression → set of IDs this expression depends on (read from)

`int a = b;` γ = { a:{b} }

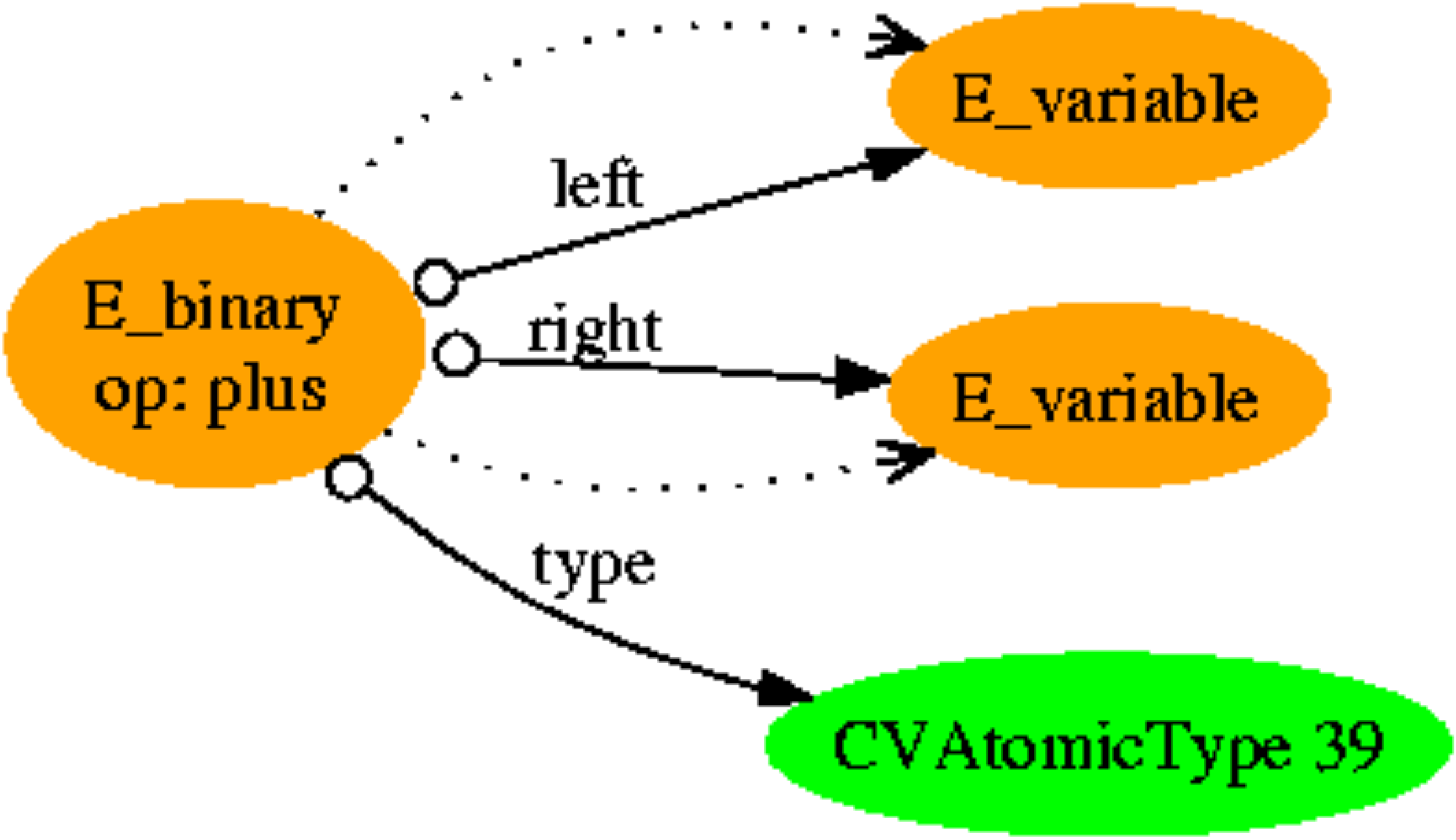
`int c = d + e;` γ = { c:{d,e} }

- Type of an expression: set of variable identifiers that **contributed** to the value of this expression

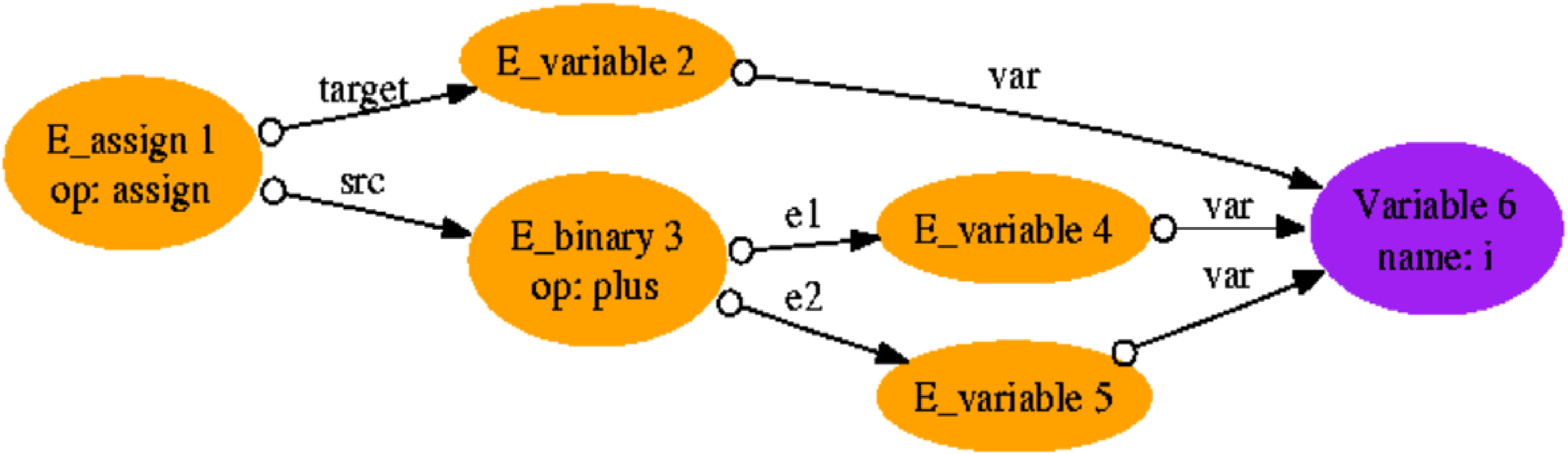
Data Flow Graph

- Transform source code of a program into an abstract syntax tree (AST)
- Traverse tree, extract data dependencies
  → simulate program run (abstract interpretation)
- Use a memory model to keep track of state changes (assignments)
- Compare inferred data flows with policy

Abstract Syntax Tree: i + j



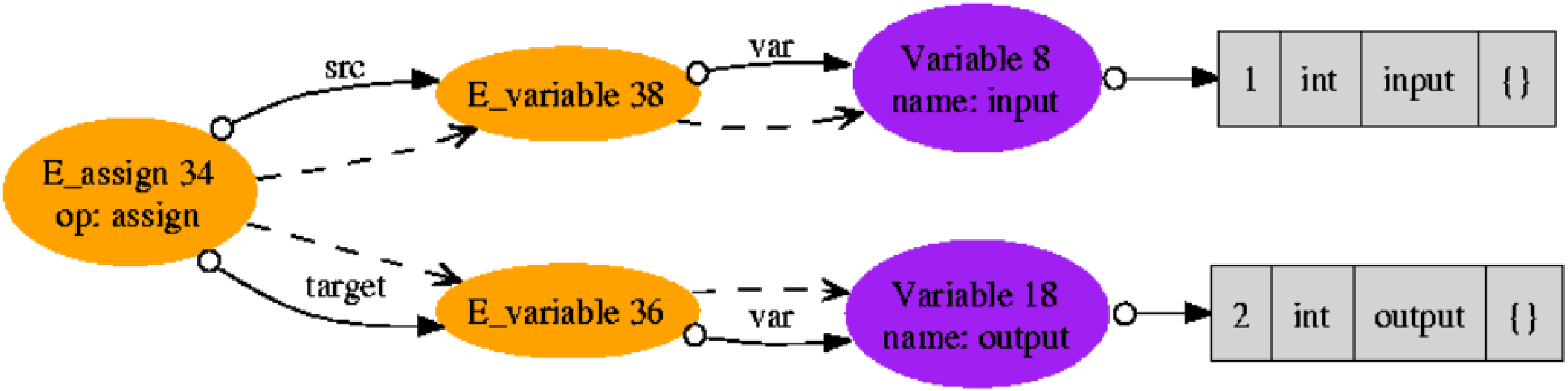
Abstract Syntax Tree: i = i + i;



- Difference between E_variable and Variable
- E_variable node refers to Variable node
- Variables keep state, are accessed (read/write) through E_variable expressions

Assignment Statements

- Information flows solely within statements
- There is no flow between statements
- Between statements data is kept in the memory (variables)



Memory Model

- Precise memory layout: not relevant
- Variables → abstract storage locations
- Infinite pool of fresh locations
- New variable declaration → new location

location	type	name	label
4711	int	i	{}

```
int i;
```

Memory Types

- Fundamental: char, int, float, bool
- Pointer/references: their type is the set of abstract locations where they point to
- References are non-NULL const pointers

location	type	name	label
1	int	i	$\{\bot_i\}$
2	float	f	$\{1,\bot_f\}$
3	ptr{1}	p	{}
4	ref{2}	g	{}

```
int i = 23;
float f;
if (i) f = 0;
int *p = &i;
float& g = f;
```

Memory Labels

- information flow type system: variables and expressions have labels describing <u>where</u> their data came from
- constants (literals) do not contain any information → modeled as ⊥

location	type	name	label
1	int	i	{}
2	int	j	{}
3	int	k	{1,2}
4	ptr{1,2}	p	{flag}

```
int i,j
int k = i + j;
int *p;
if (flag)
     p = &i;
else
     p = &j;
```

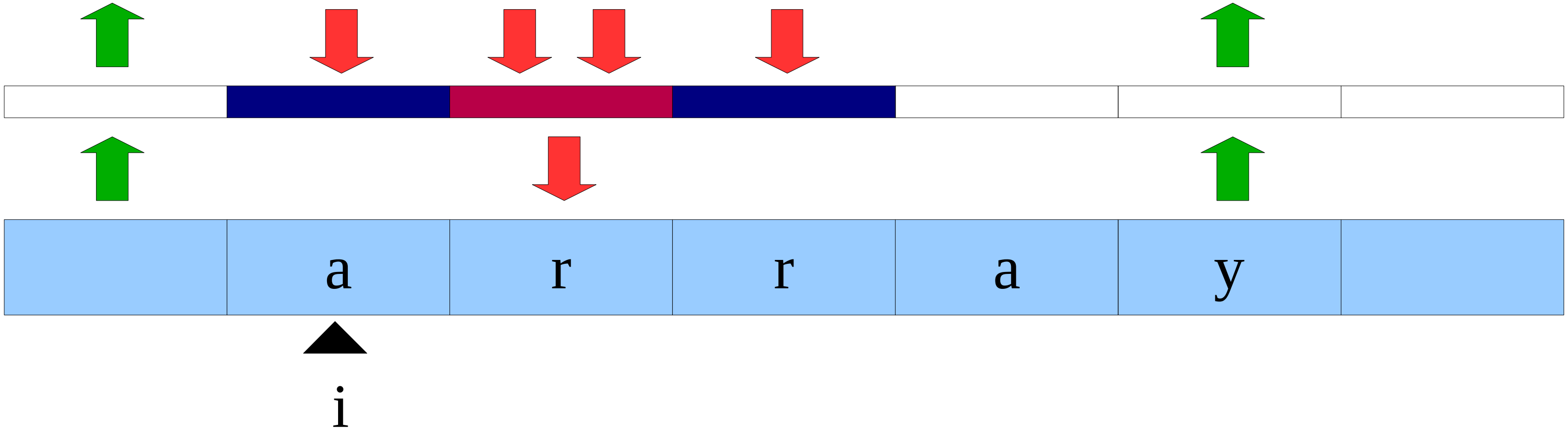
If ~ Then ~ Else

```
if (flag) do_it(); else do_something();
```

- depending on the evaluation of the condition, **either** *do_it* **or** *do_something* is executed
- therefore both run in the context of the condition (and depend on it)
- Process *then* and *else* independently
- Merge results (least upper bound)

If ~ Then ~ Else: Memory

```
if (flag) {
     array[i  ] = array[i-1];
     array[i+1] = what;
} else {
     array[i+2] = array[i+4];
     array[i+1] = ever;
}
```



Strong vs. weak updates

- Writing to **one known** memory location
  - Strong update
- Writing to **some** location of a known set
  - weak updates on all elements of this set

- Example: `array[i] = confidential`
  - If i is unknown → weak update on whole array (pessimistic estimation)
  - and assure `i >= 0 && i < max_array_index`

- Loss of information, less precise data flow graph, might cause type checking to fail

Uncertainty, Imprecision

- Why weak updates, why imprecision?
  1) Data flow analysis at compile time, inputs not (yet) available
  2) Abstract interpretation → precise values of variables are ignored

- The more precise the model is the more complex it will be (quickly far too complex)
- e.g. variable values partially modeled:
  - Ranges (for i = 0 to 9) x
  - with steps (for i = 0 to 25 step 5) $a \cdot x$
  - plus an offset (for i = 5 to 30 step 5) $a \cdot x + b$

Model Checking & Theorem Proving

- Model checking
  - Explore whole state space, reduce or cut of unfeasible paths as soon as possible
  - Example: if (flag) then ~ else → two states

- Theorem proving
  - Use (complex) formula to represent program, prove properties (e.g. array access never out of bounds → no need to check index)

- Very simple programs: done automatically
- Often: semiautomatic, interactive, guided

Does it work? SeL4!

- Third generation microkernel, based on L4, influenced by EROS, roughly 9.000 SLOC
- Formally verified
  - Systems programmer: bottom up
  - Formal methods guys: top down
  - → intermediate model in Haskell
- Start with a high level of abstraction
  - Formal specification
  - Refine model, prove correctness of refinement
  - Finally prove refinement to C-code
- No null pointer dereference, no buffer overflow, syscalls terminate, no out of kernel memory

Example: Scheduling

```
schedule =
        threads : set = get_all_ready_threads;
        thread : Thread = select threads;
        switch_to thread or switch_to_idle_thread;
```

- Pseudo code
- Very high abstraction level
  - → good for reasoning
  - → far away from actual implementation
- Make a more precise model, prove that it actually is a refinement

Refinement

```
schedule =
        prio = get_highest_priority;
        queue : Queue = get_prio_queue prio;
        thread : Thread = get_runnable_thread queue
        switch_to thread
```

- Detailed model (priorities, queues, ready state)
- Obligation: proving this model is a refinement of the former one
- Doing this iteratively → closer and closer to an implementation
- Last step: actual C-code is also a refinement

Summary

- Bell – La Padula: security levels + categories
- Type systems (const, non-Null, tainted, ...)
- Security type systems : non-interference
- HiStar and Loki: labels and tagged memory
- Data flow analysis, abstract interpretation
- Briefly: theorem proving, SeL4