

Fine Grained Access Control using Information Flow Analysis



Sruthi Bandhakavi

Information-flow : Need end-to-end guarantees

- Downloadable Tax Document Preparer:



Information-flow : Need end-to-end guarantees

- Downloadable Tax Document Preparer:



Information-flow : Need end-to-end guarantees

- Downloadable Tax Document Preparer:

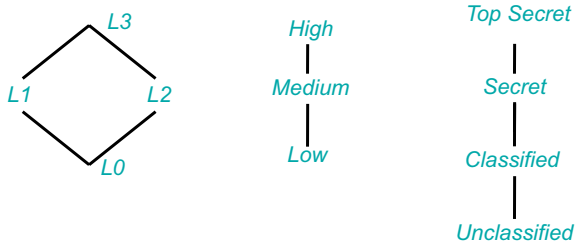


- OS level access controls don't prevent propagation of information
- In general, can't feasibly compute over encrypted data
- Need end-to-end guarantee of confidentiality

Policies are expressed as a Lattice Model

[Bell & LaPadula(1973) and Denning(1975)]

Security levels are specified in terms of a lattice (L,·).



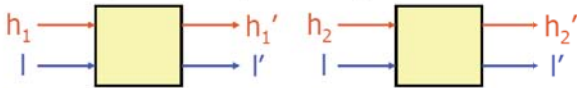
Different Kinds of Flows Considered

- Explicit Flow: Flows due to direct assignment of variables
 - Ex: $l = h$
- Implicit Flows: Flows due to control flow in the program
 - Ex: if $h = 1$ then $l = 1$ else $l = 0$;

Noninterference : A formal property for specifying valid flows (Goguen and Meseguer 1982)

- Have to check that under **all flows** of the program under **all executions** are acceptable.
- Noninterference property: For any two runs of program, Low-indistinguishable input states yield Low-indistinguishable output states.
- *Equivalently* [Cohen]: L out *independent* of initial H in.

H
↓
L

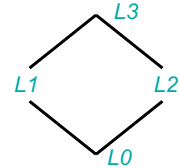


5

5

Denning's Model for information flow

- Assign a security label \underline{x} (determined statically), to every variable x .
- A program is only if has **secure flows**.
- Explicit flow: $y := x$
 - A flow $x \rightarrow y$ is **certified** iff $\underline{x} \sqsubseteq \underline{y}$
- Implicit flow: *if* $x > y$ *then* $k := 1$ *else* $j := j + 1$
 - The statement is certified if $\text{lub}(\underline{x}, \underline{y}) \sqsubseteq \text{glb}(k, j)$



6

6

Well-typed programs are Noninterferent Volpano, Smith & Irvine ('96)

- First to recognize and prove connection between type checking and non-interference.
- Each program variable is annotated with a security type along with the normal type.
- The lattice order on security levels corresponds to sub-typing relation on types.
- Program certification \equiv Type checking

H
↓
L

Insecure Programs

$l = h;$
If $h = 1$ *then* $l = 1$ *else* $l = 0;$

Secure Programs

$h = l;$
If $l = 1$ *then* $h = 1$ *else* $h = 0;$

7

7

Type Checking For Direct Flows

$h : \text{float} \quad l : \text{int}$

- **Not Allowed:** $l = h$
- **Allowed:** $h = l$



$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash e' : \tau_2 \quad \tau_2 \sqsubseteq \tau_1}{\Gamma \vdash e = e'}$$

8

8

Type Checking For Direct Flows

$h : \text{float} \quad l : \text{int}$

- **Not Allowed:** $l = h$
- **Allowed:** $h = l$



$h : \text{High} \quad l : \text{Low}$

- **Not Allowed:** $l = h$
- **Allowed:** $h = l$



$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash e' : \tau_2 \quad \tau_2 \sqsubseteq \tau_1}{\Gamma \vdash e = e'}$$

8

8

Discussion

- Separates security policies from enforcement algorithms
- Now information leaks can be checked by the compiler. Compiler produces a compilation error if it detects a leak according to the type checking rules.
- No interaction required – as easy to use as a compiler
- Fast checking – as fast as a compiler
- Noninterference is too strong : [MyersL97,...]
 - Sometimes it is required to leak a small amount of information
 - No encryption & password lookup

9

9

Need Fine-Grained Control over Info Flow

- Downloadable Tax Document Preparer:



- Owner of the data should be able to control the flow of information even in a remote system.

10

10

Need Fine-Grained Control over Info Flow

- Downloadable Tax Document Preparer:



- Owner of the data should be able to control the flow of information even in a remote system.

10

10

Decentralized Label Model [Myers & Liskov 97]

- Decentralized control
 - Fine grained information flow policies
- Declassification
 - Allows owners to downgrade the security level where necessary.
- Polymorphic Types
 - code re-use (library procedures), generic policies
- Label Inferencing
 - Local variables for which labels need not be predetermined.
- Run-time label checking
 - Flexible labels, done when labels are not known statically

11

11

Decentralized Label Model [Myers & Liskov 97]

- Decentralized control**
 - Fine grained information flow policies**
- Declassification**
 - Allows owners to downgrade the security level where necessary.**
- Polymorphic Types
 - code re-use (library procedures), generic policies
- Label Inferencing
 - Local variables for which labels need not be predetermined.
- Run-time label checking
 - Flexible labels, done when labels are not known statically

12

12

Labels

- Every Data Item has an attached label.
- A label has a set of *owners* (principals)
- For each owner, a set of *readers* (principals)
 - $\{\text{owner} : \text{reader}, \dots; \dots\}$
 - $\{\text{Bob} : \text{Bob}, \text{Preparer}; \text{Preparer} : \text{Preparer}\}$
- Each owner's policy is satisfied by ensuring that
 - effective reader set* is intersection of readers of each of the owners
 - principals may change the part they own

13

13

Explicit Flow

- Assignment *relabels* a value
 - $x := y$
 - Okay if \underline{x} is at least as restrictive as \underline{y} ($\underline{y} \sqsubseteq \underline{x}$)
- $\underline{y} \sqsubseteq \underline{x}$ means:
 - $\text{owners}(y) \subseteq \text{owners}(x)$
 - $\text{readers}(y, O) \supseteq \text{readers}(x, O)$
- The relation defines valid relabeling
 - $\{\text{Bob} : \text{Bob}, \text{Preparer}\} \sqsubseteq \{\text{Bob} : \text{Bob}; \text{Preparer} : \text{Preparer}\}$

14

14

Derived Labels

- Combining values : *Joining* labels

$$\underline{y} + \underline{z} \equiv \underline{y} \sqcup \underline{z}$$

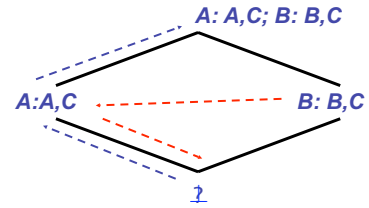
- The join of two labels (\sqcup) is the least restrictive label that is at least as restrictive as each of the labels
 - owners($L1 \sqcup L2$) = owners($L1$) \cup owners($L2$)
 - readers($L1 \sqcup L2$, O) = readers($L1$, O) \cap readers($L2$, O)

15

Policy Annotations

- Augment types with labels:

```
int{A: A, C} a;
int{B: B, C} b;
int{A: A, C ; B: B, C} c;
```



- Type-checking detects illegal flows
- Explicit Flow

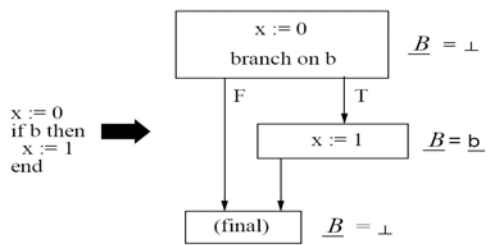
❖ a = b; // {B: B, C} $\not\sqsubseteq$ {A: A, C}

✔ c = a + b; // {A: A, C} \sqsubseteq {A: A, C ; B: B, C}

16

Implicit Flow

- Implicit Flows: A basic block label includes the labels of all values that were observed to reach that point in the execution

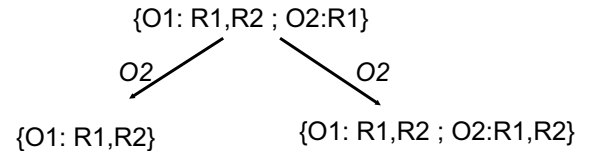


The program type-checks if $\underline{B} \sqsubseteq \underline{x}$

17

Declassification

- Label tracks policy owners
- A principal can rewrite its part of the label



- Other owners' policies still respected
- Requires test of authority (ActsFor)

18

Declassification Example: Annotated Password Checker

```
checkpassword(db:array[pinfo{chkr:chkr}]{chkr:chkr},
             user: string{client:chkr},
             password: string{client:chkr})
returns (ret:bool{client:chkr})
.....
If (db[i].names = user & db[i].passwords = password) then
    match = true ;                %{client:chkr ; chkr:chkr}
end
.....
If_acts_for(checkpassword,chkr) then
    ret := declassify(match,{client:chkr})    % ?
end
```

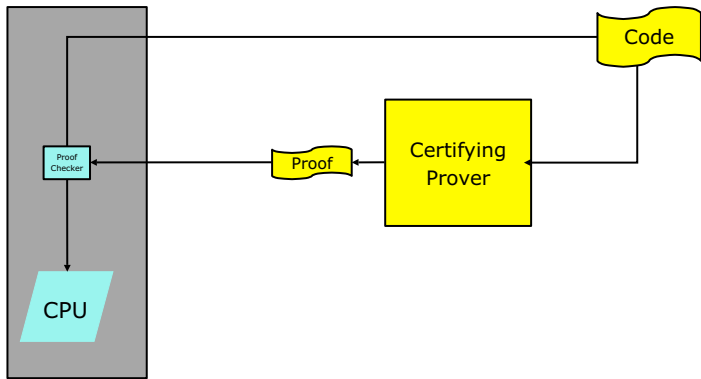
19

Applications

- Secure Program Partitioning [ZdancewicZNM01]
- Building Secure Distributed Systems [ZhengCZM03]
- Other Applications (written in Jif)
 - Implementation of cryptographic protocols [AskarovS05]
 - Jif Policy Email System(JPMail) [HicksAM06]

20

Proof Carrying Code Idea: Checkable Certificates



21

21

The Code Safety Problem*

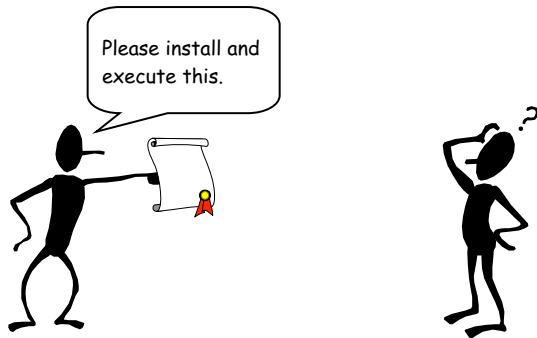


* Slides taken from Peter Lee's lecture notes on PCC

22

22

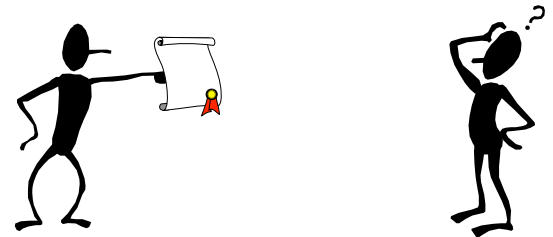
The Code Safety Problem*



* Slides taken from Peter Lee's lecture notes on PCC

22

22

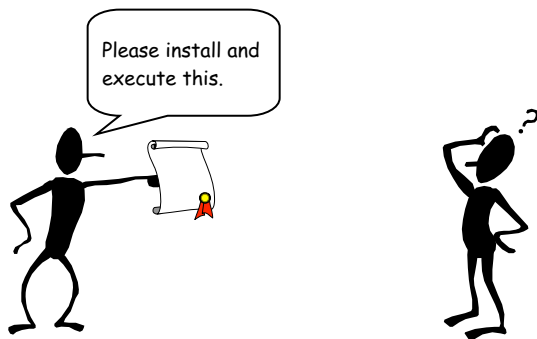


Code producer

Host

23

23

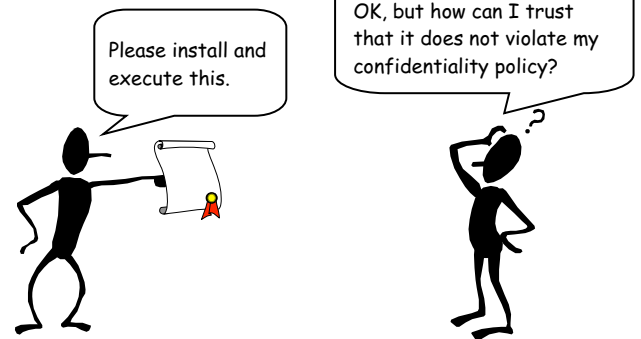


Code producer

Host

23

23



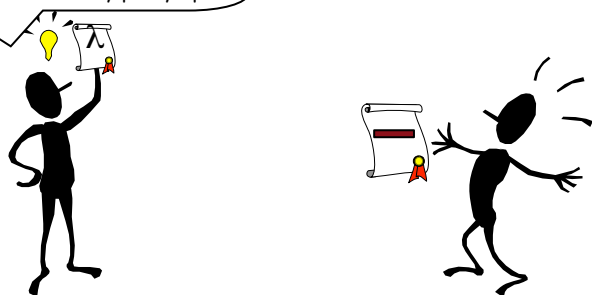
Code producer

Host

23

23

My compiler checks that the program is type safe according to the confidentiality policy C_p .



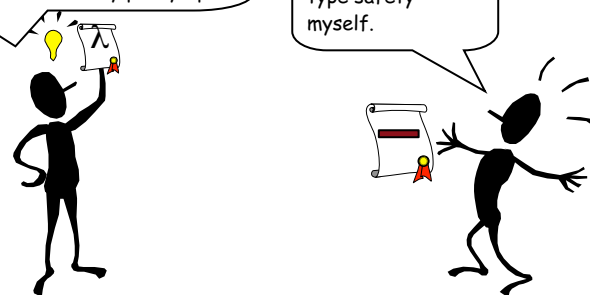
Code producer

Host

24

24

My compiler checks that the program is type safe according to the confidentiality policy C_p .



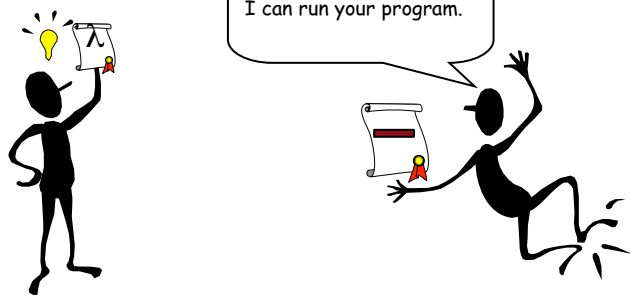
Code producer

Host

24

24

Your proof checks out. I can run your program.



Code producer

Host

25

25

Potential Limitations

- False Positives:
 - Type checking is flow-insensitive. [MizunoS92, BergerettiC85, AndrewsR80, AmtoftB03, AmtoftBB06]
Ex: $h = l$; $l = h$; as h 's type is immutable
 - Static analysis is imprecise :
Ex: if h then $l = 0$ else $l = 0$; cannot say statically which branch is taken
- Do not consider covert channels : information leaks through termination, timing, power analysis, etc.

26

26

Take Away Points

- Information Flow Analysis is a formal method to verify that programs satisfy a confidentiality or integrity policy
- A common way to prove that a program adheres to a policy is to prove that the program satisfies the *non-interference* property
- Type based information flow models
 - Provide a framework for formal verification of flow policies
 - Can catch most of the illegal information flows statically
 - Encourage programmers think of security right from the design phase

27

27

References

- DenningD77 - D. Denning and P. Denning. Certification of programs for secure information flow. *CACM* 20(7):504–513, 1977.
- GoguenM82 - J. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, 1982.
- VolpanoS196 - D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. In *Journal of Computer Security*, 4(3):167–187, 1996
- MizunoS92 - M. Mizuno and D. A. Schmidt. A security flow-control algorithm and its denotational semantics correctness proof. In *Formal Aspects of Computing*, 4(6A):727–754, 1992.
- BergerettiC85 - Jean-Francois Bergeretti and Bernard A. Carre. Information-flow and data-flow analysis of while programs. In *ACM Transactions of Programming Languages and Systems*, 7(1):37–61, Jan. 1985.
- AndrewsR80 - G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. In *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, Jan. 1980.
- ZdancewicZNM01- Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, Andrew C. Myers. Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP), Banff, Canada, pages 1–14, October 2001.

28

28

- [AmtoftB03](#) - T. Amtoft and A. Banerjee. Information flow analysis in logical form. In SAS, LNCS 3148, pages 100–115. Springer-Verlag, 2004.
- [AmtoftBB06](#) - Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In Principles of Programming Languages, 2006.
- [MyersL97](#) - A. Myers and B. Liskov. A decentralized model for information flow control. In ACM Symposium on Operating Systems Principles, pages 129–142, October 1997.
- [ZhengCZM03](#) - Lantian Zheng, Stephen Chong, Andrew C. Myers, Steve Zdancewic. Proceedings of the 2003 IEEE Symposium on Security and Privacy, Oakland, California, May 2003, pages 236–250.
- [AskarovS05](#) - A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In ESORICS, LNCS 3679, pages 197–221. Springer-Verlag, 2005.
- [HicksAM06](#) - Boniface Hicks, Kiyam Ahmadizadeh, and Patrick McDaniel. Understanding practical application development in security-typed languages. In 22st Annual Computer Security Applications Conference (ACSAC), Miami, FL, December 2006. Awarded best student paper.