Language Based Security (LBT)

Dead store Elimination and security

Chiara Bodei, Gian-Luigi Ferrari

Lecture April, 9 2025

Outline

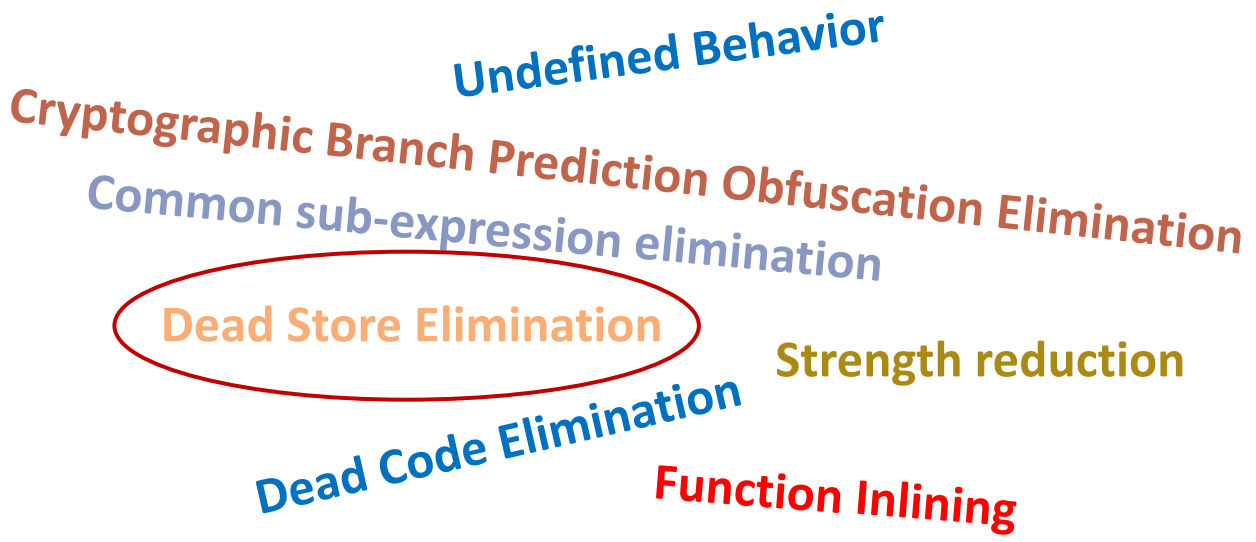
- Dead store elimination and security
- Difficulties for security validation
- Dataflow information
- Information leakage
- Hoare Logic
- Taint Proof System
- Procedure at work
- Final considerations

Compiler-induced security vulnerabilities

- Security properties should be preserved after compilation, but…
  compiler optimizations may hinder security

Vulnerabilities introduced by compiler optimizations

The compiler may **correctly** optimize source code designed for security purposes in a way that might negate the intended security guarantee



Dead store elimination

Example: **Operations that scrub memory of sensitive data**, such as passwords or cryptographic keys, can lead to **dead stores**

```
crypt(){
key = 0xC0DE; // read key
... // work with the secure key
key = 0x0; // scrub memory
}
```

Once removed the above dead store, an attacker may be able to read the key, also after the procedure has terminated

Information leaks through persistent state

- **Dead Store Elimination (DSE)** is a possible compiler optimization, built on liveness analysis
- DSE is run multiple times, as transformations may introduce further dead stores

But

- **Dead store elimination** allowed sensible data to persist more than intended
- A persistent state security violation is triggered when when data remains in memory beyond the boundaries set by the developers
- Hence, the security property exceeds the semantic functionality scope of language specifications (orthogonal w.r.t. correctness), e.g., the lifetime/region of sensitive data

Security concerns (like making sure sensitive data is wiped from memory) go beyond what programming languages are *designed* to guarantee in terms of correctness. So even if your code is "correct" according to the language rules, it might still be insecure.

- **"The security property exceeds the semantic functionality scope of language specifications"** means that protecting data (like making sure passwords or keys are wiped from memory) isn't something that most programming languages promise to handle. The language focuses on *functionality* and *correctness* — not necessarily on security.
- **"Orthogonal with respect to correctness"** is a fancy way of saying that *security* is a separate concern from *correctness*. You can write a program that is 100% correct (does everything it's supposed to), but still leaves sensitive data lying around in memory — which is a security risk.
- **The example about "lifetime/region of sensitive data"** refers to how languages manage memory. Even if you think you've deleted or overwritten data, compilers or optimizers (like dead store elimination) might have removed that part of your code, assuming it's useless. But to a security-minded person, that removal is dangerous — because it means the sensitive data could still be in memory longer than intended.

Dead store elimination

- store `x := 0` becomes skip because x is not **live**, i.e., used "after" the assignment
- P and Q have the same input-output behavior: the transformation is correct
- The **secret password is leaked** through the stack in Q: the transformation is **insecure**

Outline

- Dead store elimination and security
- Difficulties for security validation
- Dataflow information
- Information leakage
- Hoare Logic
- Taint Proof System
- Procedure at work
- Final considerations

Securing a compiler transformation

- A compiler can be correct and yet be insecure
- There are issues in dead store elimination optimizations
- There are **workarounds** which can be applied to fix problems, but they rely on programmers' awareness and knowledge of compiler's internal workings

E.g., in C if `x` is declared `volatile`, the
compiler does not remove any assignment to x: not very portable
It is a compiler specific solution: programmer awareness is required
It is also strong, it inhibits any assignment to x

↓ You should be familiar with internal working of compiler

Securing a compiler transformation *

- A possible solution is **translation validation**:
  - given an instance of a correct transformation, checks whether it is secure

Considering just a single run of the compiler at the time on the single program and not the compilation in general

Tackle an instance of a transformation and check if it is secure

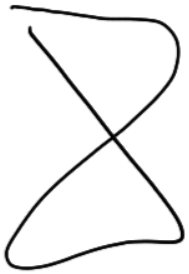
* Chaoqiang Deng, Kedar S. Namjoshi. *Securing a compiler transformation*. Formal Methods Syst. Des. 53(2), 2018.

Inherent difficulties for security validation

- How difficult is it to check the security of a transformation a posteriori?
  Given programs P, Q, and a list of eliminated stores,
  - validating correctness is in PTIME, but
  - validating security is undecidable
- Any sound validator will be incomplete

Is there a practical, provably secure dead-store elimination procedure?

- New procedure, based on taint propagation and program flow



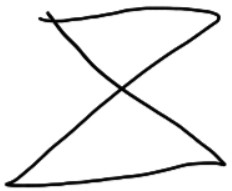
Verifying Correctness with Translation Validation

To verify the **correctness** of a DSE instance (P,Q,D), with

- P input program,
- Q resulting program, and
- D list of removed stores

- Compute dead stores D_P in P; check that $D_P \supseteq D$
- Check that P and Q have bisimilar control-flow graphs up to the labeling of edges in D

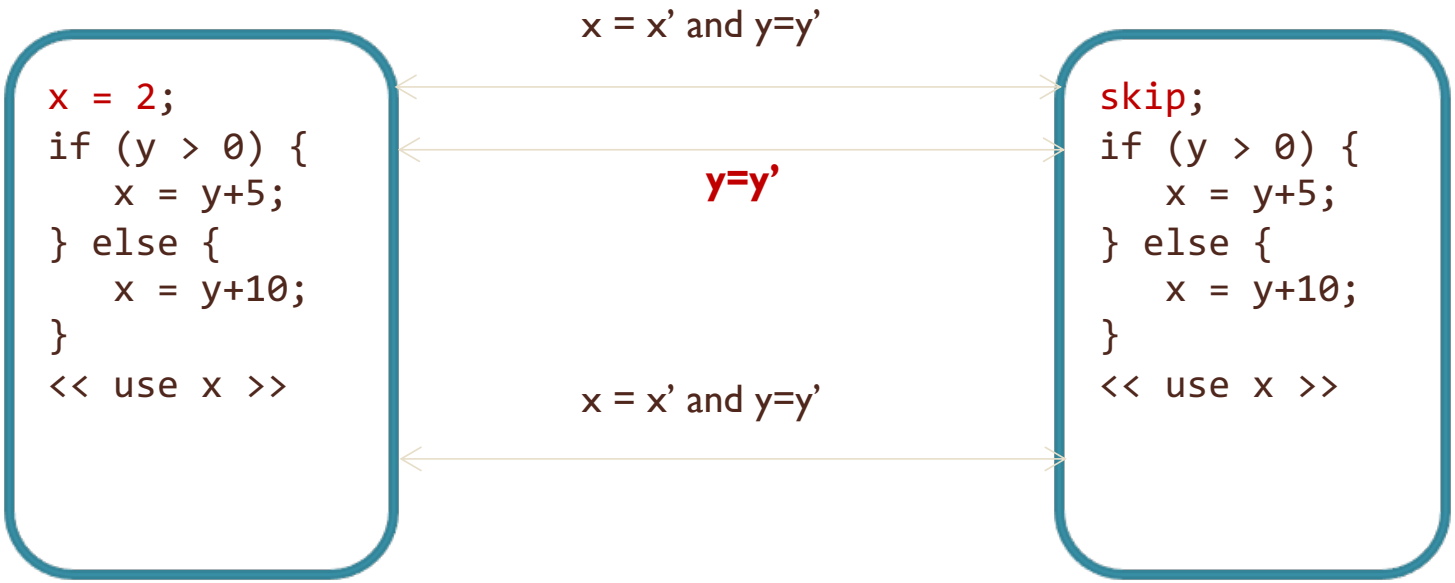
- This procedure is in PTIME



To check for correctness of transformation, you try to see if at any point the variables are more or less the same without big modifications.

Induced refinement relation

At corresponding program points p (in P) and q (in Q), every **live** variable at p has identical values at p and q



But security properties are not preserved

Verifying Correctness with Translation Validation

Validating **security** is instead undecidable

The difference between correctness and security is fundamentally due to the fact that:

- **correctness** can be defined by considering **individual executions**
- information flow requires the consideration of **pairs of executions**

- Need for a stronger notion of refinement which preserves information flow, and use it to show that several common compiler optimizations do preserve information flow properties

The procedure

- The algorithm takes as input a program P and a list of dead assignments
- It prunes that list to those assignments whose removal does not introduce a new information leak
- It removes them from P, obtaining the result program Q
- The analysis of each assignment relies on
  - control flow information from P (in the form of dominance relations) and
  - taint information (via a Hoare-style proof system)

The algorithm can be proven secure, according to a notion of secure transformation

PROCEDURE

Approach: I take a program and peform live analysis and have a list of dead assignments as a result. If I remove them I still ensure correctness.

- I have this procedure that takes the list of dead assignments and provides a sublist of dead assignments that can be safe to remove.
- So I obtain a resulting program @ and the algorythm can be proven secure. Of course I work with over approximation, so there are dead stores I could safely remove but I don't.

Our approach fuses control flow information of P(in terms of dominance relations) and taint information that propagates through execution.

Mixing these, we can understand 3 cases in which removing dead assignments is leak,

- As we said it is sub optimal because it might retain safe dead stores.
- It enforces relative security: it only ensures that no new leaks are introduced, not that leaks are removed.

Premises

The algorithm

1. is sub-optimal: it may retain more stores than is strictly necessary
2. enforces relative rather than absolute security: it does not eliminate information leaks from P, it only ensures that no new leaks are introduced in the transformation from P to Q
3. It gives guarantee only for information leakage. Other security aspects must be checked separately

It does not add leaks that were not present before

SEE UP

Example

[LIVENESS ANALYSIS]

⟦x = 2⟧= ⟦y>0⟧ \{x}

⟦y>0⟧= ⟦x = y+5⟧ U ⟦x = y+10⟧ U {y}

⟦x = y+5⟧ = ⟦<< use x >>⟧ \{x} U {y}

⟦x = y+10⟧ = ⟦<< use x >>⟧ \{x} U {y}

⟦<< use x >>⟧ = … U {x}

```
x = 2;
if (y > 0) {
    x = y+5;
} else {
    x = y+10;
}
<< use x >>
```

Example

⟦x = 2⟧= ⟦y>0⟧ \{x}
⟦y>0⟧= ⟦x = y+5⟧ U ⟦x = y+10⟧ U {y}
⟦x = y+5⟧ = ⟦<< use x >>⟧ \{x} U {y}
⟦x = y+10⟧ = ⟦<< use x >>⟧ \{x} U {y}
⟦<< use x >>⟧ = … U {x}

```
x = 2;
if (y > 0) {
    x = y+5;
} else {
    x = y+10;
}
<< use x >>
```

On every execution path:
x is redefined and the value assigned in
x = 2 is never used

Is it safe to remove?

Preliminaries: program syntax

WE WILL WORK WITH A SIMPLE IMPERATIVE LANG. TO SHOW WHAT WE NEED

- All variables have Integer type
- Variables are partitioned into input and state variables:
  - **state variables** are low security (in set **L**),
  - **input variables** may be low or high security (in sets **L** and **H**)

↗ SMALL IMP. LANGUAGE

$x \in \mathbb{X}$ — variables

$e \in \mathbb{E} ::= c \mid x \mid f(e_1, \ldots, e_n)$ — expressions: f is a function, c a constant

$g \in \mathbb{G}$ — Boolean conditions on $\mathbb{X}$

$S \in \mathbb{S} ::= \mathsf{skip} \mid \mathsf{out}(e) \mid x := e \mid S_1; S_2 \mid \mathsf{if}\ g\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \mid \mathsf{while}\ g\ \mathsf{do}\ S\ \mathsf{od}$ — statements

- A program can be represented by its **control flow graph**

We divide variables in two sets: L, H (low and high security).

📦 Input variables

These are variables that represent the **input** to your program—like user input, file input, network packets, etc. Think of them as the stuff coming into your system from the outside.

Some of these inputs are **high security** (like passwords, private keys, secret files) and some are **low security** (like public settings, non-sensitive data). That's why they're split into **L (low)** and **H (high)**.

🏠 State variables

These are more like the **internal state** of the program that gets updated and stored as it runs. They only belong to **L (low security)** because the program is *not supposed to store secrets* into these. If it does, that's a potential *leak*.

So imagine:

- `x` = a user password (H → high)
- `y` = an internal counter or log (L → low)

If a high-security input like `x` somehow ends up changing a low-security state variable like `y`, that could be a **leak of secret data**—and secure compilers are trained to freak out about that.

Preliminaries: program semantics

A **program state** s is a pair (m, p), where:

- m is a CFG node (location of s)
- p is a function mapping each variable to a value from its type (also extended to expressions)

- In the initial state, located at the entry node, state variables have a fixed valuation

Outline

- Dead store elimination and security
- Difficulties for security validation
- Dataflow information
- Information leakage
- Hoare Logic
- Taint Proof System
- Procedure at work
- Final considerations

Preliminaries: [post]-dominance

A set of nodes N in CFG **dominates** a node **m**

- if each path in the CFG from the entry node to m has to pass through at least one node of N

A set of nodes N in CFG **post-dominates** a node **m**

- if each path in the CFG from m to the exit node has to pass through at least one node of N, i.e., contains at least one node of N

- Dominators and post-dominators tell us which basic block must be executed **prior** to, of **after**, a block **m**

Data-flow problem: [post]-dominance

Domination relations are graph-theory relations and can be built using a dataflow analysis

Dominance

- **Lattice**: powerset of nodes in CFG
- At each CFG node v: $[[v]] = \{v\} \cup \bigcap_{w \in pred(v)} [[w]]$
- **Join**: set intersection

union of intersection of predecessors

Forward analysis

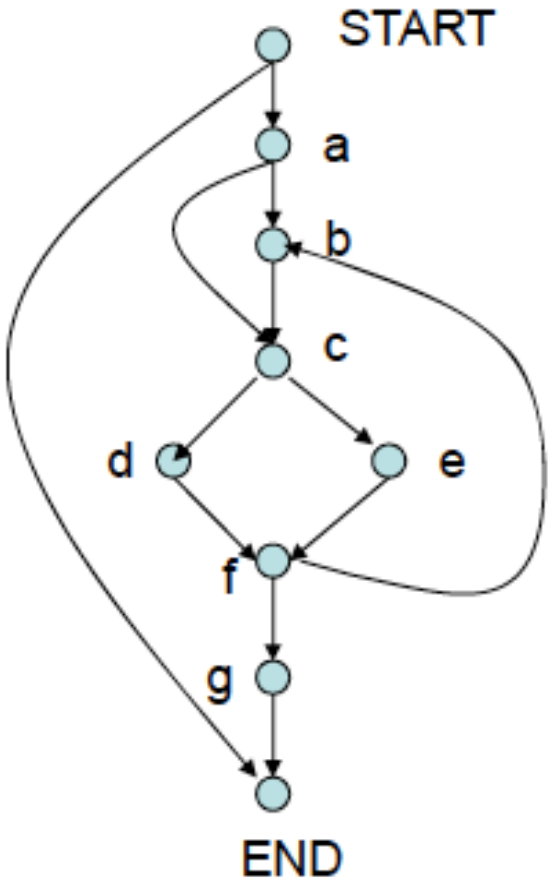
Post-dominates

- **Lattice**: powerset of nodes in CFG
- At each CFG node v: $[[v]] = \{v\} \cup \bigcap_{w \in succ(v)} [[w]]$
- **Join:**

Backward analysis

successors

Preliminaries: dominance

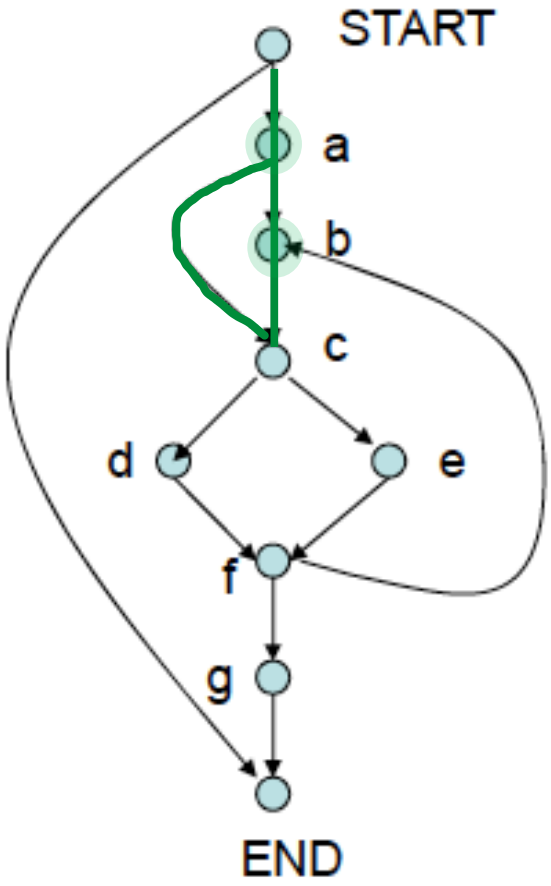


A set of nodes N in CFG **dominates** a node m

- if each path in the CFG from the entry node to m has to pass through at least one node of N

c is **dominated** by a and by b

Preliminaries: dominance

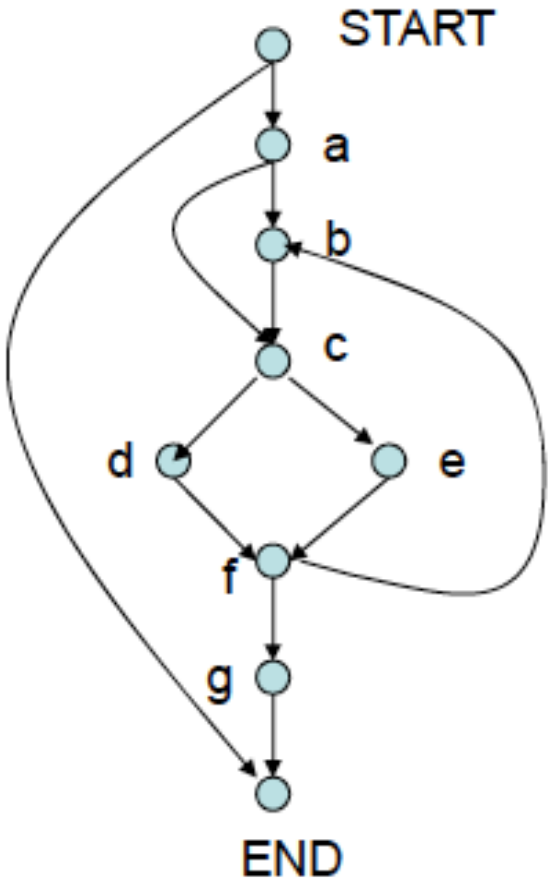


A set of nodes N in CFG **dominates** a node m

- if each path in the CFG from the entry node to m has to pass through at least one node of N

c is **dominated** by a and by b

Preliminaries: [post]-dominance



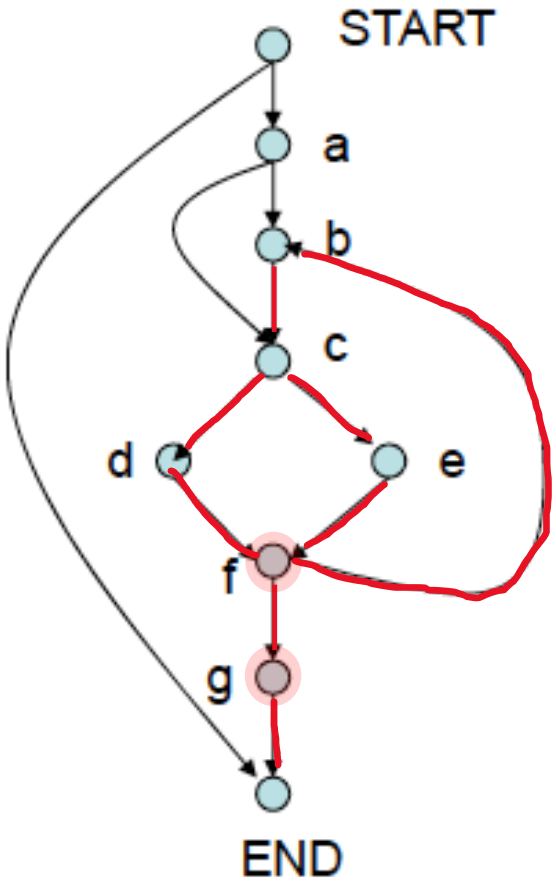
A set of nodes N in CFG **post-dominates** a node m

- if each path in the CFG from m to the exit node has to pass through at least one node of N, i.e., contains at least one node of N

c is **post-dominated** by f and by g

↑ it includes f and g. Every path

Preliminaries: [post]-dominance



A set of nodes N in CFG **post-dominates** a node m

- if each path in the CFG from m to the exit node has to pass through at least one node of N, i.e., contains at least one node of N

c is **post-dominated** by f and by g

Outline

- Dead store elimination and security
- Difficulties for security validation
- Dataflow information
- Information leakage
- Hoare Logic
- Taint Proof System
- Procedure at work
- Final considerations

Preliminaries: information leakage

Consider a deterministic program P

- Partition input variables into **Low** (L) and **High** (H) security
- All state variables are Low security

P **leaks information** if ∃ inputs (H=a,L=c) and (H=b,L=c) s.t. the resulting computations:

- differ in the sequence of output values, or
- terminate in different states (differ in the value of one of the L-variables)

We call (a, b, c) a **leaky triple** for program P

We partition variables into low and high and all state variables are low. We are going to see what "LEAKING INFORMATION" means:

"Leaking Info" means that if there are inputs in which you have a high input and an input that is low, and you change the high input, they result in different output values or they terminate in different states (differ in the value of one of the L-variables).

NOTE: information leaking is limited to the fact that high variables are not observable by attacker. Attacker might not know their value but observe their behaviour implicitly.

X

3. Definition of Leakage

The program **leaks information** if an attacker (who can observe all Low outputs and final Low variable values) can **deduce anything about the High inputs** by comparing behaviors of the program under different High inputs.

Formally, the program leaks if:

- There exist two High inputs `a` and `b`, and a shared Low input `c` (i.e., inputs `(H=a, L=c)` and `(H=b, L=c)`) such that:
  - **Either** the **sequence of output values** differs — the attacker sees something different while the program is running.
  - **Or** the **final state (i.e., Low variable values)** differs — the attacker sees a different result at the end.

Such a triple `(a, b, c)` is called a **leaky triple**, because it shows that the program's observable behavior (which the attacker can see) is influenced by a change in the High input, even though the Low input remained the same.

This implies that **information about the High (secret) input has leaked** through the Low (public) channels.

So the attacker (that cannot reach high variables, but only observe low variables) can construct a set of inputs in which a modification in the high variables of the input, but not on the low variable of the input, will provide a visible change (so in the Low channel) either through output change or through change of low variables. The attacker can use that information to then infer what the effect of high variables is, correct? A not leaky program would prevent any kind of inference because modification of high input variables doesn't have any visible effect for the attacker, correct?
Is the definition of leakage usually only given using the concept of high variables modifications?

Exactly, Giovanni — you've got it right, and you've worded it very well.

ℹ️ On the Use of High Variable Modification in Leakage Definitions

Yes — **most formal definitions of information leakage** (especially in deterministic settings) are indeed based on this **idea of varying High inputs** while **keeping Low inputs fixed**, and observing if any change occurs in the **Low-observable part** of execution.

Preliminaries: correct vs secure transformations

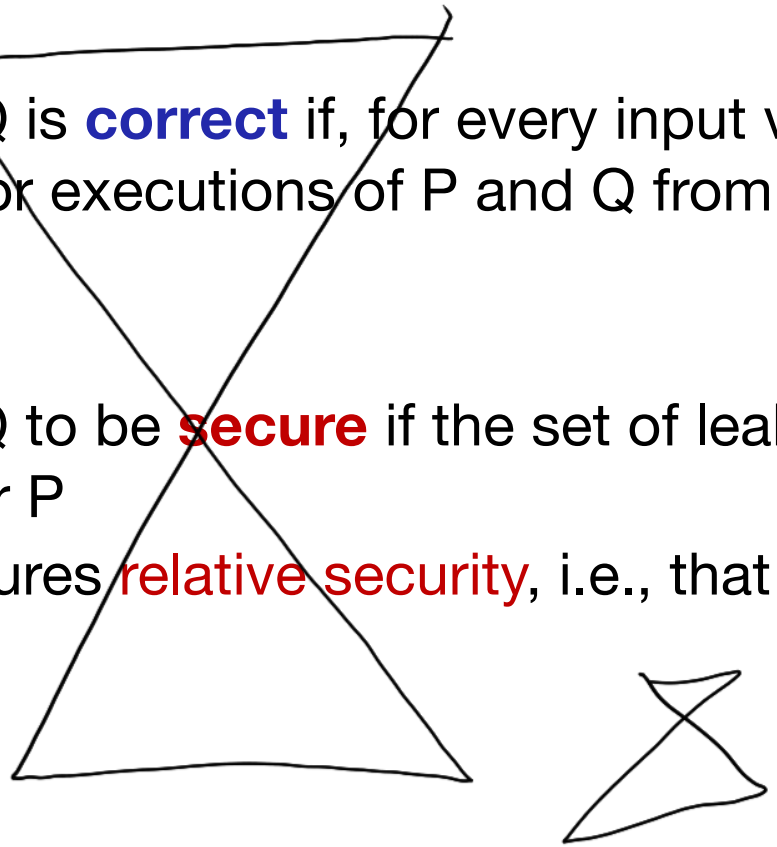
Program transformations are assumed not to alter the set of input variables, but only state variables

A transformation from P to Q is **correct** if, for every input value a, the sequence of output values for executions of P and Q from a is identical

Q is at least correct as P

A transformation from P to Q to be **secure** if the set of leaky triples for Q is a subset of the leaky triples for P

A secure transformation ensures relative security, i.e., that Q is not more leaky than P



Preliminaries: information leakage

Ex: for input values h=0, h=1, the final values of x differ

Explicit flow

```
High h
Low x initially 0

x = h;
```

```
High h
Low x initially 0

if (h > 0) {
   x = 10;
} else {
   x = 20;
}
```

Implicit flow

h: from outside you can see this condition

A transformation from P to Q is **secure** if every leaky triple for Q is also leaky for P (Q does not add any)

Outline

- Dead store elimination and security
- Difficulties for security validation
- Dataflow information
- Information leakage
- Hoare Logic → AKA Axiomatic Semantics
- Taint Proof System
- Procedure at work
- Final considerations

Preliminaries: Axiomatic semantics

- Based on formal logic, **axiomatic semantics** was introduced for formal program verification
- It defines axioms and inference rules for each language statement
- Inference rules allows one to transform expressions to other expressions
- Expressions are called assertions and state the relationships and constraints among variables that are true at a specific point in the execution

① We know operational semantics, there's also denotational semantics

② Allows you to associate logical assertions to pieces of programs.

You already know operational semantics. This allow you to associate logical assertions to pieces of code. The idea is to have these kinds of tuples (Hoare tuples) $\{P\}\ S\ \{Q\}$ in which S is a statement, P and Q are preconditions and post conditions. Idea is "what happens when I perform statement S? If I start to perform S in a state where the preconditions hold, then I should end up in a state that satisfies post conditions.

We assign to S P and Q

EX: $\{b>0\}\ a=b+1\ \{a>1\}$

WE TALK ABOUT BECAUSE IT WORKS IF S TERMINATES

- This is a "partial correctness specification", meaning that the semantics of statement S starting from a state satisfying P am included in Q (in the form of over approximation). [THIS CITATION IS UNCLEAR]

① • "If you do what the semantics of S expect you to do starting from a state that satisfy P, then you end up in state that satisfies Q".

$[[S]]P \subseteq Q$

My interpretation: when we say $\{b>0\}\ a=b+1\ \{a>1\}$, we should say: $\{b>0\}\ a=b+1\ \{a>1, b>0\}$, so we enrich the conditions. But of course conditions could change so who knows. Interpret it as nomenclature for ①

Possibility: $[[S]]P$ means applying S to state satisfying P, that will lead you in states that satisfy Q [over approximation, Q might be larger] → can include non reachable states

→ No false negatives, all executions of a verified behavior behave correctly

JMP §3

Axiomatic semantics

- Before a statement we have pre-conditions and after post-conditions

$$\{P\}\ S\ \{Q\}$$

Hoare triple: S started in any state satisfying P will satisfy Q on termination

- Example:

$$\{b>0\}\ a = b+1\{a>1\}$$

Hoare triples

Precondition

$$\{P\}\ S\{Q\}$$

Postcondition

Partial correctness specification Semantics of this statement starting with if S is executed in a state where P is true, and the execution terminates with success, then Q is guaranteed to be true afterwards:

$$[[S]]P \subseteq Q$$

can include non reachable states

Semantics of S

over approximation

Semantics of statement: if you do what they expect you to do, starting from states that satisfy P, you end up in state that satisfy Q.

No false negatives: all executions of a verified program behave correctly

Use of Hoare triples

A triple $\{P\}\ c\ \{Q\}$ can be seen in different ways

If you start from there, understanding what can be put there is a way to

- **Semantics**: Given P and c, determine Q such that $\{P\}\ c\ \{Q\}$ is a way to describe the behaviour of c

USEFUL FOR

$$\{x = 1\}\ x := x + 1\ \{x = 2\}$$

- There are several uses of those Hoare triples: $\{P\}\ c\ \{Q\}$

① Semantics meaning: Given P and c, determine Q such that $\{P\}\ c\ \{Q\}$ is a way to describe the behavior of c. Understanding the post condition is the way to describe behavior of C.

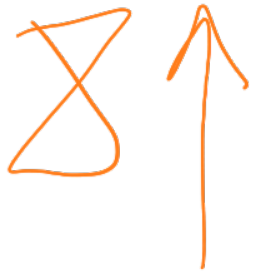
② Specification purpose: Given P and Q, finding c such that $\{P\}\ c\ \{Q\}$ means writing the program that realizes what specified by pre and post conditions say.

③ Correctness: Given a triple P, c, Q, you can prove that it is correct, meaning that it is true that semantics of command c applied in states that satisfy the precondition is included in states that satisfy post conditions.

JMP 3 slides

Use of Hoare triples

A triple $\{P\}\ c\ \{Q\}$ can be seen in different ways



USEFUL FOR

- **Specification**: Given P and Q, determine c, such that $\{P\}\ c\ \{Q\}$, means Given P and Q, determine c, such that $\{P\}\ c\ \{Q\}$, means to write the program that realizes what specified by the pre and postcondition given

$$\{x = 1\}\ x := x + 1\ \{x = 2\}$$

Use of Hoare triples

A triple $\{P\}\ c\ \{Q\}$ can be seen in different ways

- **Correctness**: Given P, c and Q, prove that $\{P\}\ c\ \{Q\}$ is correct corresponds to a verification of correctness of c w.r.t the specification determined by P and Q

$$\{x = 1\}\ x := x + 1\ \{x = 2\}$$

Use of Hoare triples

A triple $\{P\}\ c\ \{Q\}$ can be seen in different ways

- **Semantics**: Given P and c, determine Q such that $\{P\}\ c\ \{Q\}$ is a way to describe the behaviour of c
- **Specification**: Given P and Q, determine c, such that $\{P\}\ c\ \{Q\}$, means Given P and Q, determine c, such that $\{P\}\ c\ \{Q\}$, means to write the program that realizes what specified by the pre and postcondition given
- **Correctness**: Given P, c and Q, prove that $\{P\}\ c\ \{Q\}$ is correct corresponds to a verification of correctness of c w.r.t the specification determined by P and Q

$$\{x = 1\}\, x := x + 1\, \{x = 2\}$$

Weaken the preconditions

Example:

$$\{\quad\}\ a = b+1\ \{a>1\}$$

- One essential precondition needed to ensure that $a>1$ is $b>0$
- Note that $b > 10$ will also guarantee that $a>1$, but this is a stronger condition
- The **weaker precondition** is **better** because it is **less restrictive** of the possible starting values of b that ensure correctness.
- Typically, given a postcondition, we would like to know the **weakest precondition** that guarantees that the program satisfies that postcondition

Axiomatic semantics (cont.)

Suppose to have the incomplete triple

$$\{?\}\ a = b+1\ \{a>1\}$$

After the assignment we want “a>1”

Intuitively, which is the condition on b that makes the post-condition verified?

Axiomatic semantics (cont.)

Suppose to have the incomplete triple

{?} a = b+1 {a>1}

After the assignment we want “a>1”

Intuitively, which is the condition on b that makes the post-condition verified?

- {b>10} can make the job, and also {b>5} can do it

Axiomatic semantics (cont.)

Suppose to have the incomplete triple

{?} a = b+1 {a>1}

After the assignment we want “a>1”

Intuitively, which is the condition on b that makes the post-condition verified?

- {b>10} can make the job, and also {b>5} can do it
- But {b>0} is the best one:

{b>10} => {b>5} => {b>0}

Requiring {b>0} is enough to guarantee that “a>1” is true after the assignment: it is the least restrictive requirement

Axiomatic program proofs

- Start from the last post-condition
- work back to the first statement:
- if the first pre-condition coincide with the program specification, the program is correct

[So what we need to do is start from the end and try to understand what is what you have at the beginning in order to obtain what you have to the end.] [UNCLEAR]

- To prove that a program is correct, you start from the last post condition and work back to the first statement. If the first pre-conditions can code with the program specification then it is correct.

Axiomatic program proofs

Example

$$\{?\}\ a = b+1\ \{a>1\}$$

- Which is the procedure to find ??
- It is not just guessing as before
- Start from the last post-condition: {a>1} and work back:
- Since a>1 must hold after the assignment, this means that b+1>1 and this can be true only if b>0
- Technically this amounts to substituting b+1 to a in a>1

$$\{b>0\}\ a = b+1\ \{a>1\}$$

Proof System

To construct formal proofs of partial correctness specifications, axioms and rules of inference are needed

The proof rules constitute an axiomatic semantics of our programming language

↓ The axioms and rules you have

🧩 1. Axiomatic Semantics

This refers to a **formal system** that assigns **logical meaning** to programs by expressing what must be true **before and after** execution of a program or a piece of code.

- It does this using **Hoare triples** of the form:

$$\{P\}\, C\, \{Q\}$$

  where:

  - `P` is the **precondition** (what we assume to be true before `C` runs),
  - `C` is a **command** (some code),
  - `Q` is the **postcondition** (what we guarantee is true if `C` terminates).

So, saying a **Hoare triple is valid** means: *if the precondition holds, and the command* `C` *terminates, then the postcondition will hold.*

This is what we call **partial correctness**: we reason about correctness **only if the program terminates**.

🧩 2. Axioms and Rules of Inference

- These are like **building blocks** or **rules** that allow us to **formally derive** valid Hoare triples.
- For example:
  - The **assignment axiom** lets us reason about what happens when a variable is assigned a value.
  - The **sequence rule** lets us reason about a block like `C1; C2`.
  - The **conditional rule** helps with `if-then-else`.

These rules collectively make up a **proof system** — the set of formal tools we use to construct logical proofs about program behavior.

So the slide is saying:

> To reason formally (prove) that our program is correct (in terms of Hoare triples), we need a collection of such rules — and these form the axiomatic semantics of the language.

When we talk about **logical proofs about program behavior**, we mean a **step-by-step derivation**, using **formal rules**, that shows a certain **property** holds for a program.

These proofs look and behave like **mathematical proofs**, but they're applied to code.

In this context:

- The **statements** you prove are **Hoare triples** like:

  $$\{P\}\ C\ \{Q\}$$

  which means: *if the precondition* `P` *holds before running command* `C`*, and* `C` *terminates, then* `Q` *will be true afterward.*

- The **proof** is a structured sequence of applications of **axioms** and **inference rules**, like:
  - The **assignment axiom**
  - The **sequence rule**
  - The **conditional rule**
  - Etc.

Each rule has a formal shape, and you apply them just like logical inference in math, chaining them together

An axiom for skip

$$\frac{}{\{P\}\ \text{skip}\ \{P\}}\ \text{AXIOM, no premise}$$

$$\{x > 0\}\ \text{skip}\ \{x > 0\}$$

An axiom for assignment

Backward oriented

$$\{assign\}\ \frac{}{\{\ ?\ \}x := a\{Q\}}$$

What do we need before the assignment so that Q holds afterwards?

Syntax replacement

Perform backward reasoning: what do we need before assignment such that Q holds formally?

$\{?\}\ x := x+1\ \{x = 2\}$ → Here new x is old x+1. So $\{x=1\}$

Rule for assignment: we will perform the backwards reasoning we were doing before.
Question: what is the precondition for assignment so that Q holds afterwards?

$$\frac{}{\{\ ?\ \}\ X := a\ \{Q\}}$$

⇓

Formally, we obtain preconditions by taking post conditions and substituting the new value of X where you had the old value of X.

WE SUBSTITUTE OCCURRENCIES OF X WITH A

$$\frac{}{\{Q[a/x]\}\ X := a\ \{Q\}}$$

↳ Formal way to write that

$\{?\}\ X = X+1\ \{X=2\}$

$\{X+1=2\} \equiv \{X=1\}\ X = X+1\ \{X=2\}$

This is true for a general expression:

$$\frac{}{\{Q[E/x]\}\ X := E\ \{Q\}}$$

- What we need now is a rule for composition (a sequential rule):

$$\frac{\{P\}\ C_1\ \{R\} \quad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1; C_2\ \{Q\}}$$

- Intuition: we see the intermediate conditions $\{R\}$, which are the post conditions of C_1 and the preconditions of C_2. You go back from C_2, solve it to find R and then keep going back for P. You start from post conditions and move and move back.

START FROM THE LAST POST CONDITION AND WORK BACK

An axiom for assignment

Backward reasoning

$$\{assign\} \quad \frac{}{\{Q[a/x]\}x := a\{Q\}}$$

What do we need before the assignment so that Q holds afterwards?

Syntax replacement

$$\{x = 1\} \equiv \{x + 1 = 2\} \equiv \{x = 2[x+1/x]\}\ x := x+1\ \{x = 2\}$$

Formally obtained by textually post conditions and substituting new value of x instead of old value.

An axiom for assignment

Backward reasoning

Start from the post-condition and work back

$$\{assign\} \quad \frac{}{\{Q[a/x]\}x := a\{Q\}}$$

What do we need before the assignment so that Q holds afterwards?

Syntax replacement

$$\{x = 1\} \equiv \{x + 1 = 2\} \equiv \{x = 2[x+1/x]\}\ x := x+1\ \{x = 2\}$$

$$\{true\} \equiv \{x = x\} \equiv \{x = x + 0y\}\ r := x\ \{x = r + 0y\}$$
$$\{x = r\} \equiv \{x = r + 0y\}\ q := 0\ \{x = r + qy\}$$
$$\{x = r + qy\} \equiv \{x = r - y + (q + 1)y\}\ r := r - y\ \{x = r + (q + 1)y\}$$

Assignment rule

An axiom for assignment

$$\frac{}{\{Q[E/x]\}\ x := E\ \{Q\}}$$

Substitute Expn. for x.

Syntax replacement

Example

{?} a = b+1 {a>1}

- Start from the last post-condition: {a>1} and work back:
  - Substitute E (b+1) for every x (a) in Q (a>1)
  - ? = Q[E/x] = (a>1)[b+1/a] = b+1 >1
  - Then ? = b>0
- Undo the assignment and solve:

{b>0} a = b+1 {a>1}

Sequential rule

$$\text{\{seq\}}\ \frac{\{P\}\ c_1\ \{R\}\quad \{R\}\ c_2\ \{Q\}}{\{P\}\ c_1; c_2\ \{Q\}}$$

Backward reasoning

Start from the last post-condition and work back

You come out with some desired post condition that becomes the precondition of the second one

$$\frac{\{x = r + qy\}\ r := r - y\ \{x = r + (q+1)y\}\quad \{x = r + (q+1)y\}\ q := q + 1\ \{x = r + qy\}}{\{x = r + qy\}\ r := r - y; q := q + 1\ \{x = r + qy\}}$$

Assignment rule

Example

$$\{?\}\ y = 3*x + 1;\ x = y + 3\ \{x<10\}$$

- Start from the last post-condition: {x<10} and work back:
  - Since x<10 must hold after the assignment, this means that y + 3 < 10 and this can be true only if y<7
  - And then? We need an intermediate condition between the two assignments: y<7
  - We further work back:

$$\{?\}\ y = 3*x + 1\ \{y<7\}$$

  - Since y<7 must hold after the assignment, this means that 3*x + 1 < 10 and this can be true only if x<2

$$\mathbf{\{x<2\}}\ y = 3*x + 1;\ x = y + 3\ \{x<10\}$$

Sequential rule

Formally:

$$\text{\{seq\}}\ \frac{\{P\}\ c_1\ \{R\}\quad \{R\}\ c_2\ \{Q\}}{\{P\}\ c_1; c_2\ \{Q\}}$$

{?} y = 3*x + 1; x = y + 3 {x<10}

- {?} x = y + 3 {x<10}
  {x<10[y+3/x]} x = y + 3 {x<10} ➡ {y<7} x = y + 3 {x<10}
- {?} y = 3*x + 1 {y<7}
  {y< 7[3*x+1/y]} y = 3*x + 1 {y<7} ➡ {x<2} y = 3*x + 1 {y<7}

{x<2} y = 3*x + 1; x = y + 3 {x<10}

Inlining

At each point before/after/in between statements, what do we know about the state of the program?

Proofs of program correctness:

- can be broken down into a few main steps, guided by the structure of the program, and
- can then be presented through a program annotated with inlined assertions

$\{x = r + qy\}$
$r := r - y;$
$\{x = r + (q + 1)y\}$
$q := q + 1;$
$\{x = r + qy\}$

look down

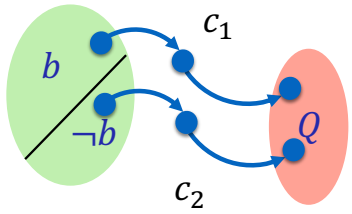
- Annotate each point i with a predicate A_i
- Successive annotations must be inductive
  $[[c_i]]\ A_i \subseteq A_{i+1}$
- $\{P\} \Rightarrow A_1$ and $A_n \Rightarrow \{Q\}$

$$\frac{\{x = r + qy\}\ r := r - y\ \{x = r + (q + 1)y\} \quad \{x = r + (q + 1)y\}\ q := q + 1\ \{x = r + qy\}}{\{x = r + qy\}\ r := r - y; q := q + 1\ \{x = r + qy\}}$$

Another thing that you can do: these are the rules and you can enforce the rules in your program. You can put the preconditions and postconditions in every point of a program. And this is a way to prove correctness in general. When you write a program, this is a way to prove it does what it is supposed to do. And wrap your head about enforcing conditions in between statements. That's it.

IF rule

$$\text{\{cond\}}\quad \frac{\{P \wedge b\}\ c_1\ \{Q\} \qquad \{P \wedge \neg b\}\ c_2\ \{Q\}}{\{P\}\ \text{if } b \text{ then } c_1 \text{ else } c_2\ \{Q\}}$$



state that satisfies condition

$\{true\} \equiv$

if $x \geq 0$ do

$\{x \geq 0\} \equiv \{x - 1 \geq 0\}$

skip;

else

$\{\neg(x \geq 0)\} \equiv \{(-x < 0)\}$

$x := -x;$

$\{x > 0\} \Rightarrow \{x \geq 0\}$

$\{x \geq 0\}$

What do we know immediately after entering the if branch?

- b is true, or we have entered the if branch
- P is true, because we have not executed any code that modifies it

look down

There is also a rule for conditions: just divide the space in states that satisfy the conditions and states that do not:

$$\frac{\{P \wedge b\}\, c_1\, \{Q\} \qquad \{P \wedge \neg b\}\, c_2\, \{Q\}}{\{P\}\ \text{if } b \text{ then } c_1 \text{ else } c_2\ \{Q\}}$$

Why `P ∧ b` and `P ∧ ¬b`?

Well, `b` is a **boolean expression**, right? So depending on the state, it might be true or false. The only thing we know *before* entering the `if` is that `P` holds. So we prepare for **both** cases:

- If `b` turns out true: we execute `c1`. We need to know that `Q` will still hold at the end, *assuming we started in a state satisfying* `P ∧ b`.
- If `b` turns out false: we execute `c2`. So we need to ensure `Q` still holds, *assuming we started in a state satisfying* `P ∧ ¬b`.

Together, these cover *all possible executions* of the `if`. We're not saying both branches will be taken at the same time—we're just proving that **no matter which one is taken**, it will work out.

Okay so this means: We have to find postconditions Q that will hold no matter which branch we choose, correct?

Exactly, Giovanni! You've got it now.

- There's also an important while rule in this analysis. Idea is that is important to find an invariant P, something that holds at each iteration.

$$\frac{\{P \wedge b\}\, c\, \{P\}}{\{P\}\ \text{while } b \text{ do } c\ \{P \wedge \neg b\}}$$

The While Rule (Hoare Logic)

You're looking at this:

$$\frac{\{P \wedge b\}\, c\, \{P\}}{\{P\}\ \text{while } b \text{ do } c\ \{P \wedge \neg b\}}$$

This is called the **Hoare rule for while-loops**. It says the following:

> If assuming `P` and `b` hold before the loop body `c`, and after executing `c` you get `P` again, then the loop `while b do c` preserves `P` when it exits — and at that point, `b` must be false.

Let's break this down properly.

🟦 `P` **is called a loop invariant**

This means:

> It holds **before the loop starts**, and it continues to hold **after every iteration** of the loop.

That's why in the **top part** of the rule, we check:

$$\{P \wedge b\}\ c\ \{P\}$$

This means: *if the invariant* `P` *holds and the loop condition* `b` *is true (so the loop executes), then after executing* `c`, `P` *must still hold.*

That's **key** — it ensures that every iteration preserves the invariant.

🟧 **The bottom part says:**

$$\{P\}\ \text{while}\ b\ \text{do}\ c\ \{P \wedge \neg b\}$$

Which means:

> If the invariant holds before the loop, and each iteration maintains it, then when the loop finishes (i.e., when `b` is false), `P` still holds, and now `b` must be false — that is, `P ∧ ¬b`.

🎯 What Does "Goal: Find an Invariant P" Mean?

In practice, when you're given a loop and asked to **prove something about it**, the **main challenge** is often to **figure out what** `P` **should be**.

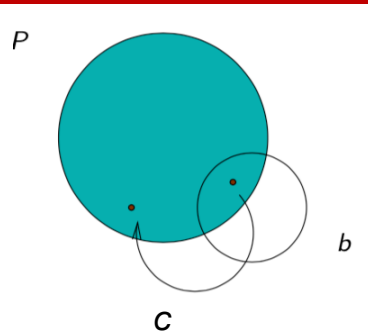
So this note:

> *"Goal: find an invariant* `P` *implied by the while precondition"*
> means:
>
> Given what you know **before the loop starts**, and what you want to be true **after the loop ends**, you must come up with a `P` that:
>
> - "Holds at the beginning,"
> - "Is preserved through each iteration,"
> - "And is strong enough to help you prove the postcondition at the end."

WHILE rule

Loop invariant: it should be true **before** and **after** each iteration of the loop body

$$\text{\{while\}}\quad \frac{\{P \wedge b\}\ c\ \{P\}}{\{P\}\ \text{while } b \text{ do } c\ \{P \wedge \neg b\}}$$



Goal: find an invariant P implied by the while precondition

At the end the condition must be false

$\{x \geq 0\}$
while $x > 0$ do
$\{x \geq 0 \wedge x > 0\} \equiv \{x - 1 \geq 0\}$
$x := x - 1;$
$\{x \geq 0\}$
$\{x \geq 0 \wedge x \leq 0\} \equiv \{x = 0\}$

The triple must be correct if the iterations are 0, 1, 2,..

Not many details. It is important to find an invariant: something that holds for each iteration

- At each point of execution, you have an invariant

Consequence rule

Precondition strengthening

{cons}

$$\frac{P \Rightarrow P' \quad \{P'\}\ c\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ c\ \{Q\}}$$

Postcondition weakening:

If we can prove Q', we can always relax it to something weaker

$P \Rightarrow P' \equiv P \subseteq P'$

The rule of consequence allows reasoning in terms of the semantics of the assertions

$$\{x \geq 0 \wedge y > 0\} \Rightarrow$$
$$\{-y < 0 \wedge x \geq 0 \wedge y \geq 0\} \Rightarrow$$
$$\{x - y < x \wedge x + y \geq 0\}$$
$$n := x - y;$$
$$\{n < x \wedge x + y \geq 0\}$$

- Consequence rule

$$\frac{P \Rightarrow P' \quad \{P'\}\, c\, \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\, c\, \{Q\}}$$

📜 The Consequence Rule

You're looking at this:

$$\frac{P \Rightarrow P' \quad \{P'\}\, c\, \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\, c\, \{Q\}}$$

What this says in words:

> If you can show that:
>
> - "your actual precondition `P` **implies** (is stronger than) the precondition `P'` you used in a proof,"
> - "and you've already proved that running `c` from `P'` ends in `Q'`,"
> - "and `Q'` **implies** (is stronger than) the postcondition `Q` you want to claim,"
>
> Then, you are allowed to conclude:

$$\{P\}\, c\, \{Q\}$$

🔍 What Does This Really Mean?

🔹 ***Precondition Strengthening*** (`P ⇒ P'`):

You're allowed to **assume more** in your proof than you actually require from the caller.
So if you can prove something works under assumption `P'`, then it also works under **any stronger assumption** `P`. [A stronger assumption implies P']

> Example: If your proof assumes `x > 10` (`P'`), and you know `x > 100` (`P`), then you're fine, because `x > 100 ⇒ x > 10`.

You're giving **more information** to your proof than what you strictly needed — that's **strengthening**.

🔹 ***Postcondition Weakening*** (`Q' ⇒ Q`):

If your proof gives you a **strong result** `Q'`, you're allowed to **settle** for something **weaker** `Q`.

> Example: If your proof shows `x = 5` (`Q'`), then you can also conclude `x ≥ 0` (`Q`) — that's weaker but still correct.

This is handy when your proof yields very specific results, but you only care about a **general property**.

You can assume MORE THAN REQUIRED (strengthen preconditions) or conclude LESS THAN PROVEN (weaker postconditions)

Weakest precondition

Given $\{R\}\ c\ \{Q\}$

EXAMPLES

it may happen that R is not the only valid precondition

Ex.

$\{x+1>0\}$	$\{x+1>30\}$	$\{x+1>100\}$	
$x = x + 1;$	$x = x + 1;$	$x = x + 1;$	…
$\{x>0\}$ ①	$\{x>0\}$	$\{x>0\}$	

Which one do you prefer?

① This is the weakest solution

Weakest precondition

Given $\{R\}\ c\ \{Q\}$
it may happen that R is not the only valid precondition

Ex.

$\{x + 1 > 0\}$
$x = x + 1;$
$\{x > 0\}$

$\{x + 1 > 30\}$
$x = x + 1;$
$\{x > 0\}$

$\{x + 1 > 100\}$
$x = x + 1;$
$\{x > 0\}$

…

Which one do you prefer? The first. Why?

Weakest precondition

Given $\{R\}\ c\ \{Q\}$
it may happen that R is not the only valid precondition

Ex.

$\{x+1>0\}$	$\{x+1>30\}$	$\{x+1>100\}$	
$x = x + 1;$	$x = x + 1;$	$x = x + 1;$	…
$\{x>0\}$	$\{x>0\}$	$\{x>0\}$	

Which one do you prefer? The first. Why?
Because it guarantees correctness for the largest set of inputs

$$\{x \mid x+1>30\} \subseteq \{x \mid x+1>0\}$$
$$\Rightarrow$$

I can strengthen the condition

Weakest precondition

$$\{ \quad \} \, x := 3 \, \{ x + y > 0 \}$$

What is the most general value of y such that $(x + y > 0)$?

Weakest precondition

$$\{y > -3\}\, x := 3\, \{\, x + y > 0 \,\}$$

What is the most general value of y such that $(x + y > 0)$?

$$(y > -3)$$

Strongest postconditions

Given $\{R\}\ c\ \{Q\}$
it may happen that Q is not the only valid precondition

Ex.

$\{x = 5\}$
$x = x + 2;$
$\{x > 0\}$

$\{x = 5\}$
$x = x + 2;$
$\{x = 7\}$

More precise

...

Which one do you prefer? The second!
Why? Because it guarantees correctness for the smallest set of inputs $\{x \mid x = 7\} \subseteq \{x \mid x > 0\}$
$\Rightarrow$

Axiomatic semantics (cont.)

QUIZ:

What is the strongest possible assertion?

And the weakest?

Axiomatic semantics (cont.)

QUIZ:

What is the strongest possible assertion? **False**

And the weakest? **True**

satisfied only by empty set:
no state satisfies false

HL proof system

$$\{\text{skip}\}\ \frac{}{\{P\}\ \text{skip}\ \{P\}} \qquad \{\text{assign}\}\ \frac{}{\{Q[a/x]\}x := a\{Q\}}$$

$$\{\text{seq}\}\ \frac{\{P\}\ c_1\ \{R\}\quad \{R\}\ c_2\ \{Q\}}{\{P\}\ c_1; c_2\ \{Q\}} \qquad \{\text{cons}\}\ \frac{P \Rightarrow P' \quad \{P'\}\ c\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ c\ \{Q\}}$$

$$\{\text{cond}\}\ \frac{\{P \wedge b\}\ c_1\ \{Q\} \qquad \{P \wedge \neg b\}\ c_2\ \{Q\}}{\{P\}\ \text{if } b \text{ then } c_1 \text{ else } c_2\ \{Q\}}$$

$$\{\text{while}\}\ \frac{\{P \wedge b\}\ c\ \{P\}}{\{P\}\ \text{while } b \text{ do } c\ \{P \wedge \neg b\}}$$

Partial Correctness

Theorem Any derivable HL triple is sound ①

↑ it approximates the semantics

Proof: By induction on the derivation tree

① This is a sort of static analysis: you work on statements to approximate behaviors. Result provided by triple that is valid, in accordance with the semantic of the statement applied from P will bring you to Q.

COMPARING WHAT YOU OBTAIN WITH BEHAVIOR GIVES YOU THAT BEHAVIOR (SOUNDNESS ALSO SEEN BEFORE)

📌 "Any derivable HL triple is sound"

This is the **soundness theorem** of Hoare Logic (HL), and it says:

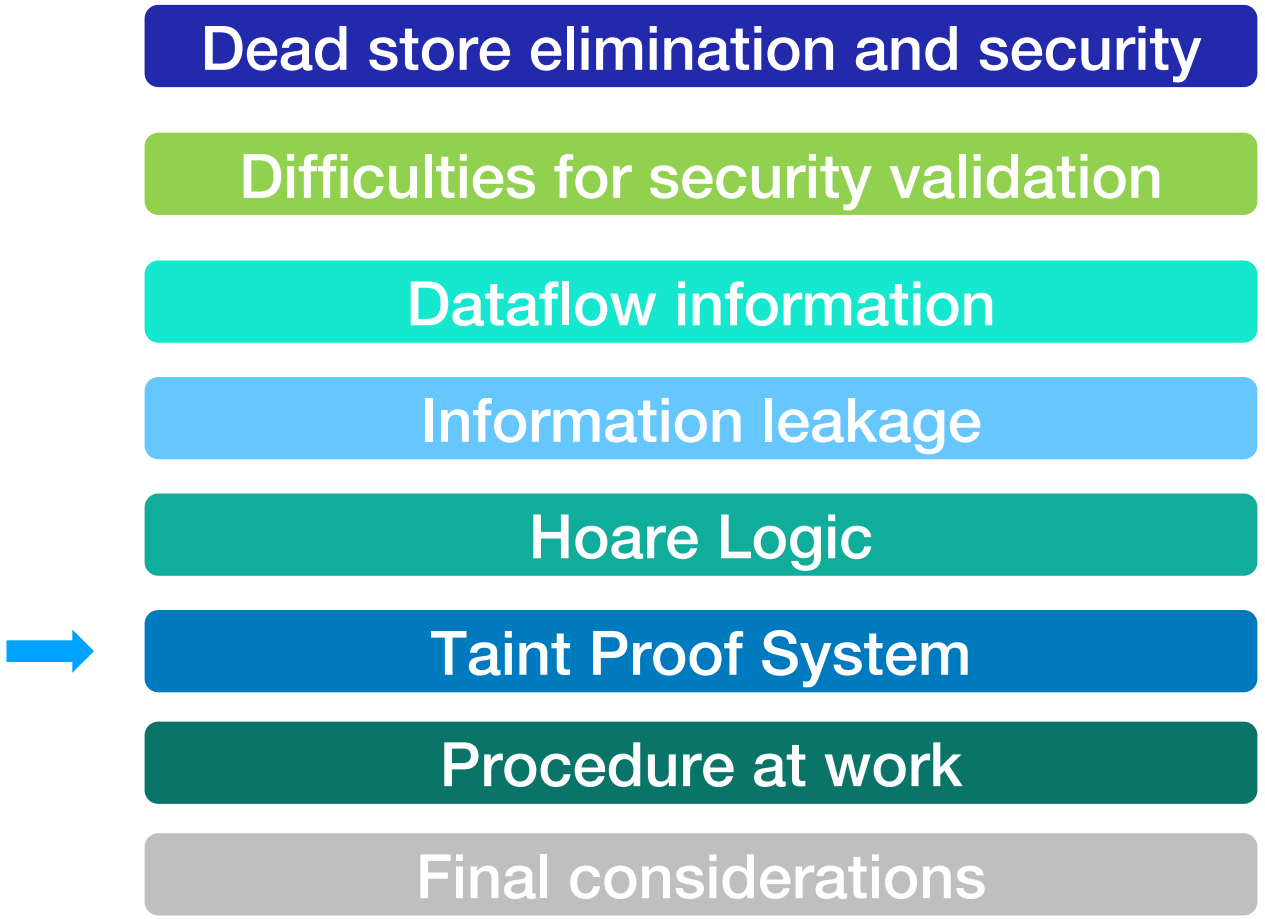
> If you can derive a Hoare triple `{P} c {Q}` using the Hoare proof system (that is, the rules we've been discussing),
> then this triple is **actually true** according to the **semantics** of the language — meaning:
>
> **Whenever** command `c` starts in a state where `P` holds,
> and `c` **terminates**,
> then the resulting state will satisfy `Q`.

In short:

> **Everything you can prove in Hoare logic is correct** — it matches the actual behavior of the program.

This is what "soundness" always means in logic:
→ **You never prove something that isn't true.**

Outline


Dead store elimination and security
Difficulties for security validation
Dataflow information
Information leakage
Hoare Logic
Taint Proof System
Procedure at work
Final considerations

A Taint Proof System

A Taint Proof System aims at tracking the influence of input variables on program state

Taint = {tainted [true], untainted [false]}

Tainted environment

It associates a taint value to each program variable name

$\mathcal{E}$: Variables -> Taint, s.t. $\mathcal{E}(x)$ = true if x is tainted

A **program state** s is a pair (m, p)

- m is a CFG node and
- p is a function mapping each variable to a value from its type

A pair of states (s,t), satisfies a tainted environment $\mathcal{E}$,

(s,t) |= $\mathcal{E}$ [with s = (m,p), t=(n,q)] if

- m=n (same location) and
- s and t have identical values for every variable x that is untainted in $\mathcal{E}$

- What can we do now? We are now interested in taints associated to variables. We divide variables in tainted ones and untainted ones. Idea is to have a tainted environment: each variable has an associated tainted and untainted label (true - false). If I take a pair of states, they satisfy a tainted environment if the following conditions are met:

🧠 Understanding Program States

In this model, a **program state** is a pair:

- `s = (m, p)`
  - `m` is a **control flow graph (CFG) node** — so it tells you *where in the program* the execution is.
  - `p` is a function mapping each variable to its value at that point in time.

So for example:

- If variable `x` has value 42, then `p(x) = 42`.

Two states, `s = (m, p)` and `t = (n, q)` can be **compared** in terms of how different they are under taint tracking.

🧩 Satisfaction of a Tainted Environment: `(s, t) ⊨ E`

"|=" : consequence logic [UNCLEAR]

Now this is the key part. We want to say whether two program states `s` and `t` are **equivalent under a taint environment** `E`.

Definition says: A pair of states satisfies a tainted environment:
`(s, t) ⊨ E` if:

1. `m = n` → the program is at the same **control point** in both states (i.e., both executions are paused at the same command).
2. For every variable `x` where `E(x) = false` (so `x` is untainted), we require that `p(x) = q(x)` (i.e., `x` has the **same value** in both states).

- So we are comparing two possible states in terms of what is tainted and what is not.

When we talk about **satisfaction of a tainted environment** — `(s, t) ⊨ E` — we are essentially comparing **two program states** `s` and `t` with respect to the **taint status** of variables in the environment `E`.

Here's why we say "equivalent under a taint environment":

🧠 What Does "Equivalent Under a Taint Environment" Mean?

We're asking: Do **both program states** (`s` and `t`) behave the same way in terms of **tainted vs. untainted variables**?

A Taint Proof System (Hoare-style)

Only interested in tainted values

$$\{\mathcal{E}\}\ S\ \{\mathcal{F}\}$$

- $\forall$ s,t: (s,t) $\models \mathcal{E} \wedge$ (s -> s') $\wedge$ (t -> t'): (s',t') $\models \mathcal{F}$
- For any pair of states that satisfy $\mathcal{E}$,
  after executing S, their successors, satisfy $\mathcal{E}$

Properties

- $\mathcal{E} \sqsubseteq \mathcal{F}$ iff $\forall$x. $\mathcal{E}$(x) implies $\mathcal{F}$(x) [monotonicity]
- $\mathcal{E}' \sqsubseteq \mathcal{E}$, $\{\mathcal{E}\}$ S $\{\mathcal{F}\}$, $\mathcal{F} \sqsubseteq \mathcal{F}'$ implies $\{\mathcal{E}'\}$ S $\{\mathcal{F}'\}$ [widening]

So here we rephrase our triple in terms of "I'm not interested in values but only in variant statuses". Similarly to the Hoare Logic principle, where is that for any pair of states that satisfy the precondition E, after the statement S, they have to satisfy F, the resulting postcondition.

$$\{E\}\,S\,\{F\}$$

That's it.

A Taint Proof System (Hoare-style)

$\mathcal{E}(c)$ gives if c is tainted

$$\mathcal{E}(c) = \text{false, if } c \text{ is a constant}$$
$$\mathcal{E}(x) = \mathcal{E}(x), \text{ if } x \text{ is a variable}$$
$$\mathcal{E}(f(t_1, \ldots, t_N)) = \bigvee_{i=1}^{N} \mathcal{E}(t_i)$$

S is skip: $\{\mathcal{E}\}\ \text{skip}\ \{\mathcal{E}\}$

S is out(e): $\{\mathcal{E}\}\ \text{out}(e)\ \{\mathcal{E}\}$

S is $x := e$: $$\frac{\mathcal{F}(x) = \mathcal{E}(e) \qquad \forall y \neq x : \mathcal{F}(y) = \mathcal{E}(y)}{\{\mathcal{E}\}\ x := e\ \{\mathcal{F}\}}$$

Sequence: $$\frac{\{\mathcal{E}\}\ S_1\ \{\mathcal{G}\} \qquad \{\mathcal{G}\}\ S_2\ \{\mathcal{F}\}}{\{\mathcal{E}\}\ S_1; S_2\ \{\mathcal{F}\}}$$

Remember $\mathcal{E}(\cdot)$ gives us a lookup in the environment. Works like the Hoare system that we have seen, but remember rule for assignment and rule for conditional.

Taint Proof System: focus on assignment rule

Rule for assignment.

$$S \text{ is } x := e: \quad \frac{\mathcal{F}(x) = \mathcal{E}(e) \qquad \forall y \neq x : \mathcal{F}(y) = \mathcal{E}(y)}{\{\mathcal{E}\}\, x := e\, \{\mathcal{F}\}}$$

The mapping for the assigned variable is changed according to the taint type of the right hand side expression

$$\frac{}{\{Q[E/x]\}\ x := E\ \{Q\}}$$

$$\mathcal{E} = \mathcal{F}[e/x]$$

$$\frac{}{\{\mathcal{F}[e/x]\}\ x := e\ \{\mathcal{F}\}}$$

x inherits the taint label of e

x inherits taint value of e.

Just the intuition: x inherits the exact value of e.

Assignment rule: examples

$$\frac{}{\{\mathcal{F}[e/x]\}\ x := e\ \{\mathcal{F}\}}$$

Examples

- {x:U, y:U} x = 0; {x:U, y:U}

  the tag of x **directly** depends on the tag of 0, while the tag of y does not change

- {x:U, y:U} x = read_password(); {x:T, y:U}

  the tag of x **directly** depends on the tag of read_password();

If you start with a situation of both x and y unbinded and assign x to 0, you have the same binding values. While if you assign read_password() that gives a binded expression, x takes (unfortunately) the binding status. SKIP ODD RULE

A Taint Proof System: conditional and loop

Conditional: For a statement S, we use $Assign(S)$ to represent a set of variables which over-approximates those variables assigned to in S. The following two cases are used to infer $\{\mathcal{E}\}\ S\ \{\mathcal{F}\}$ for a conditional:

c untainted

$$\text{Case A: } \frac{\mathcal{E}(c) = \text{false} \qquad \{\mathcal{E}\}\ S_1\ \{\mathcal{F}\} \qquad \{\mathcal{E}\}\ S_2\ \{\mathcal{F}\}}{\{\mathcal{E}\}\ \text{if } c \text{ then } S_1 \text{ else } S_2 \text{ fi}\ \{\mathcal{F}\}}$$

c tainted

$$\text{Case B: } \frac{\mathcal{E}(c) = \text{true}^{①} \qquad \{\mathcal{E}\}\ S_1\ \{\mathcal{F}\} \qquad \{\mathcal{E}\}\ S_2\ \{\mathcal{F}\} \qquad \forall x \in Assign(S_1) \cup Assign(S_2) : \mathcal{F}(x) = \text{true}}{\{\mathcal{E}\}\ \text{if } c \text{ then } S_1 \text{ else } S_2 \text{ fi}\ \{\mathcal{F}\}}$$

$$\text{While Loop: } \frac{\mathcal{E} \sqsubseteq \mathcal{I} \qquad \{\mathcal{I}\}\ \text{if } c \text{ then } S \text{ else skip fi}\ \{\mathcal{I}\} \qquad \mathcal{I} \sqsubseteq \mathcal{F}}{\{\mathcal{E}\}\ \text{while } c \text{ do } S \text{ od}\ \{\mathcal{F}\}}$$

① Condition might be tainted.

- Case in which value used to evaluate condition is tainted (might be under the control of the attacker). Under this rule, you have:

$$\frac{\mathcal{E}(c) = \text{true} \quad \{\mathcal{E}\}\, S_1\, \{F\} \quad \{\mathcal{E}\}\, S_2\, \{F\} \quad \forall x \in \text{Assign}(S_1) \cup \text{Assign}(S_2) : F(x) = \text{True}}{\{\mathcal{E}\}\ \text{if } c \text{ then } S_1 \text{ else } S_2\ \{F\}}$$

→ set of variables that over approximates the ones assigned to in S_1

You set as tainted all the variables

If condition is tainted, then condition propagates to all the variables used in your branches. For instance if c is tainted, x is now tainted.

FALSE CASE:

$$\frac{\mathcal{E}(c) = \text{false} \quad \{\mathcal{E}\}\, S_1\, \{F\} \quad \{\mathcal{E}\}\, S_2\, \{F\}}{\{\mathcal{E}\}\ \text{if } c \text{ then } S_1 \text{ else } S_2\ \{F\}}$$

Conditional rule: focus on case B

c tainted

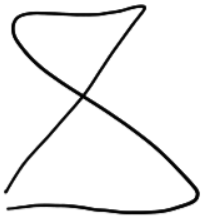
$$\frac{\mathcal{E}(c) = \text{true} \quad \{\mathcal{E}\}\ S_1\ \{\mathcal{F}\} \quad \{\mathcal{E}\}\ S_2\ \{\mathcal{F}\} \qquad \forall\ x \text{ in } \text{Assign}(S_1)\ \cup\ \text{Assign}(S_2):\ \mathcal{F}(x)}{\{\mathcal{E}\}\ \text{if c then } S_1 \text{ else } S_2\ \{\mathcal{F}\}}$$

If the condition is tainted each variable assigned in the condition branches becomes tainted

Example

- {c:T, x:U, y:U} if c then x = y else x = z {c:T, x:T, y:U}

  the tag of x indirectly depends on the tag of c

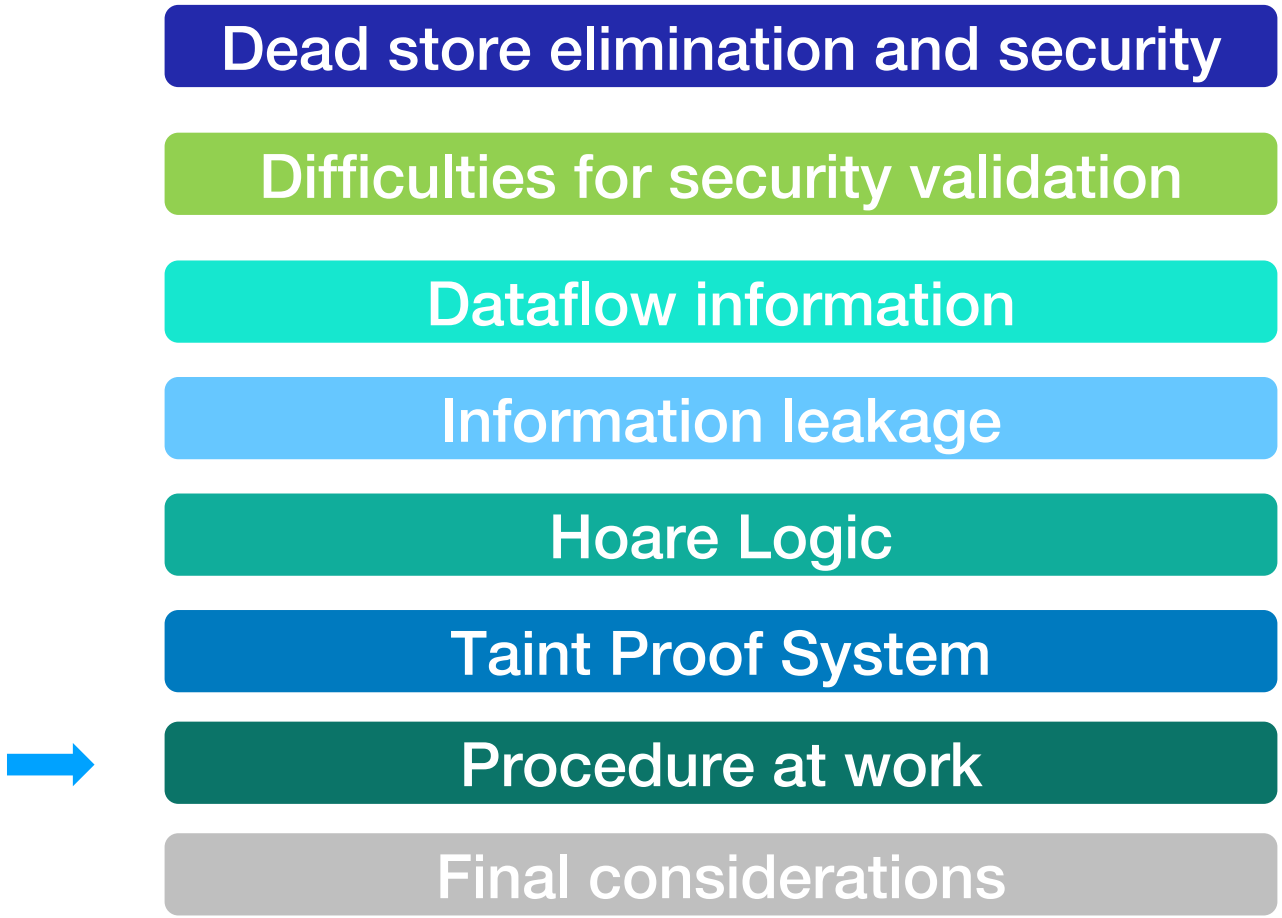


Taint analysis

The proof system is sound

Theorem 3 (Soundness) *Consider a structured program P with a proof of $\{\mathcal{E}\}\ P\ \{\mathcal{F}\}$. For all initial states (s, t) such that $(s, t) \models \mathcal{E}$: if $s \xrightarrow{P} s'$ and $t \xrightarrow{P} t'$, then $(s', t') \models \mathcal{F}$.*

Outline


Dead store elimination and security
Difficulties for security validation
Dataflow information
Information leakage
Hoare Logic
Taint Proof System
Procedure at work
Final considerations

The algorithm for calculating taints

The proof system can be turned into an algorithm for **calculating taints**

- the proof rule for each statement other than the while can be read as a monotone forward environment transformer
- for while loops, the proof rule requires the construction of an inductive environment, $\mathcal{I}$. This can be done through a least fixpoint calculation for $\mathcal{I}$ based on the transformer for the body of the loop
- The entire process is thus in **polynomial time**

We have info on how nodes dominate others, we know what info leakage happens (a high input changing that modifies behaviour), we can understand which dead stores are safe to remove.

The proof system we have defined can be turned in an algorithm that calculates taints in every point of our program. This can be done in polinomial time. Now let's jump in the example. We have inference for taint pre-post conditions.

Taint analysis at work

```
int foo()
{
   int x,y;

   x = 0;

   y = read_user_id();
   if (is_valid(y)) {
      x = read_password();

      login(y,x);
         x = 0;

   } else {
      printf ("Invalid ID");
   }

   return;
}
```

Taint analysis at work

```
int foo()
{
   int x,y; // everything is untainted
   {x:U,y:U}
   x = 0;

   y = read_user_id();
   if (is_valid(y)) {
      x = read_password ();

      login(y,x);
         x = 0;

   } else {
      printf ("Invalid ID");
   }

   return;
}
```

Taint analysis at work

```
int foo()
{
   int x,y; // everything is untainted
   {x:U,y:U}
   x = 0;

   y = read_user_id();
   if (is_valid(y)) {
      x = read_password();

      login(y,x);
         x = 0;

   } else {
      printf ("Invalid ID");
   }

   return;
}
```

Taint analysis at work

```
int foo()
{
   int x,y; // everything is untainted
   {x:U,y:U}
   x = 0; // tag of x depends on tag of 0
   {x:U,y:U}
   y = read_user_id();
   if (is_valid(y)) {
      x = read_password();

      login(y,x);
         x = 0;

   } else {
      printf ("Invalid ID");
   }

   return;
}
```

Taint analysis at work

```
int foo()
{
   int x,y; // everything is untainted
   {x:U,y:U}
   x = 0; // tag of x depends on tag of 0
   {x:U,y:U}
   y = read_user_id();
   if (is_valid(y)) {
      x = read_password();

      login(y,x);
         x = 0;

   } else {
      printf ("Invalid ID");
   }

   return;
}
```

Taint analysis at work

```
int foo()
{
   int x,y; // everything is untainted
   {x:U,y:U}
   x = 0; // tag of x depends on tag of 0
   {x:U,y:U}
   y = U: read_user_id();
   if (is_valid(y)) {
      x = read_password();

      login(y,x);
         x = 0;

   } else {
      printf ("Invalid ID");
   }

   return;
}
```

Taint analysis at work

```
int foo()
{
   int x,y; // everything is untainted
   {x:U,y:U}
   x = 0; // tag of x depends on tag of 0
   {x:U,y:U}
   y = U: read_user_id();
   if (is_valid(y)) {
      x = read_password();
      login(y,x);
         x = 0;

   } else {
      printf ("Invalid ID");
   }

   return;
}
```

Can be dangerous, if x is returned

Taint analysis at work

```
int foo()
{
   int x,y; // everything is untainted
   {x:U,y:U}
   x = 0; // tag of x depends on tag of 0
   {x:U,y:U}
   y = U: read_user_id();
   if (is_valid(y)) {
      x = T: read_password();//change
      {x:T,y:U}
      login(y,x);
         x = 0;

   } else {
      printf ("Invalid ID");
   }

   return;
}
```

Taint analysis at work

```
int foo()
{
   int x,y; // everything is untainted
   {x:U,y:U}
   x = 0; // tag of x depends on tag of 0
   {x:U,y:U}
   y = U: read_user_id();
   if (is_valid(y)) {
      x = T: read_password();//change
      {x:T,y:U}
      login(y,x);
         x = 0;

   } else {
      printf ("Invalid ID");
   }

   return;
}
```

Taint analysis at work

```
int foo()
{
   int x,y; // everything is untainted
   {x:U,y:U}
   x = 0; // tag of x depends on tag of 0
   {x:U,y:U}
   y = U: read_user_id();
   if (is_valid(y)) {
      x = T: read_password();//change
      {x:T,y:U}
      login(y,x);
         x = 0; //tag of x is untainted again
         {x:U,y:U}
   } else {
      printf ("Invalid ID");
   }

   return;
}
```

Taint analysis at work

```
int foo ()
{
  int x,y; // everything is untainted
  {x:U,y:U}
  x = 0; // tag of x depends on tag of 0
  {x:U,y:U}
  y = U: read_user_id();
  if (is_valid(y)) {
    x = T: read_password();//change
    {x:T,y:U}
    login(y,x);
      x = 0; //tag of x is untainted again
      {x:T,y:U}
  } else {
    printf ("Invalid ID");
  }
  {x:U,y:U}
  return;
}
```

Taint analysis at work

```
int foo()
{
   int x,y; // everything is untainted
   {x:U,y:U}
   x = 0; // tag of x is untainted
   {x:U,y:U}
   y = U: read_user_id();
   if (is_valid(y)) {
      x = T: read_password();//change
      {x:T,y:U}
      login(y,x);
         x = 0; //tag of x is untainted again
         {x:T,y:U}
   } else {
      printf ("Invalid ID");
   }
   return;
}
```

Dead Store

Dead Store

Secure DSE procedure

- The algorithm
  - takes a program P and a list of dead assignments, then
  - prunes that list to those assignments whose removal is guaranteed not to introduce a new information leak.
  - This is done by consulting the result of a control-flow sensitive taint analysis on the source program P and exploiting post-dominance relations
- As the algorithm removes a subset of the known dead stores, the transformation is correct
- It is possible to prove that it is also secure

So, the algorithm takes a program and a list of dead assignments, then prunes that list to one whose assignments' removal is guaranteed not to introduce a new information leak. This is done using information given by the taint analysis and the dominance information we have about nodes.

The idea is:

You analyze the dead stores in your list, and they can be removed harmlessly in at least one of these three possible situations:

1. The store is post-dominated by other stores
2. variable x is untainted before the store and untainted at the exit from the program
3. variable x is untainted before the store, other stores to x are unreachable, and this store post dominates the entry node.

Secure DSE procedure

1. Compute the control flow graph G for the source program S
2. Set each internal variable at the initial location as Untainted, each L-input as Untainted, and each H-input as Tainted
3. Do a taint analysis on G
4. Do a liveness analysis on G and obtain the set of dead assignments, DEAD
5. **while** DEAD *is not empty* **do**
   - Remove an assignment, A, from DEAD, suppose it is "$x := e$"
   - Let CURRENT be the set of all assignments to x in G except A
   - **if** A *is post-dominated by* CURRENT **then** [Case 1]
     - Replace A with skip
     - Update the taint analysis for G
   - **else if** x *is Untainted at the location immediately before* A *and* x *is Untainted at the final location of* G **then** [Case 2]
     - Replace A with skip
   - **else if** x *is Untainted at the location immediately before* A *and there is no path from* A *to* CURRENT *and* A *post-dominates the entry node* **then** [Case 3]
     - Replace A with skip
   - **else**
     - (* Do nothing *)
   - **end**
   - **end**
6. Output the result as program T

Condition 1

Condition 2

Condition 3

Secure DSE procedure

Consider a candidate dead store to variable x

It may be removed if:

- At least

1) the store is post-dominated by other stores

any leak through x must arise from the dominating stores

2) variable x is untainted before the store and untainted at the exit from the program

the taint proof is unchanged, so a leak cannot arise from x; other flows are preserved

3) variable x is untainted before the store, other stores to x are unreachable, and this store post-dominates the entry node

- the taint proof is unchanged, so a leak cannot arise from x; other flows are preserved

Case 1: post-domination

Every path to the exit from the first assignment, x = 0, passes through the second assignment to x. It can be safely removed

```
void foo()
{
     int x;
     {x:U}
     x = T:read_password();
     {x:T}
     x = U:0;// Dead Store
     {x:U}
     x = U:5;// Dead Store
     {x:U}
     return;
}
```

Post-dominated by the next assignment

sat Case 1

The two dead assignments are redundant

Every path to the exit from the 1st assignment passes through the 2nd assignment

Removing the first dead assignment is secure as the remaining assignment blocks the password from being leaked outside the function

The obtained program has the same CFG

Case 2: stable untainted assignment

Variable x is untainted before the dead store and is untainted at the program exit

```
int foo()
{
     int x, y;
     {x:U,y:U}
     x = 0;  // Dead Store
     {x:U,y:U}
     y = U:read_user_id();
     if (is_valid(y)) {
          x = T:read_password ();
          {x:T,y:U}
          login(y,x);
          x = 0;    // Dead Store
          {x:U,y:U}
     } else {
          printf ("Invalid ID");
     }
     {x:U,y:U}
     return;
}
```

x is untainted before and after the assignment

sat Case 2

This removal is secure, as x remains untainted at the final location:
hence, it does not leak information about the password

Case 3: final assignment

The second assignment is always the final one and the variable x is untainted before and the store post-dominates the entry node

```
void foo()
{
     int x, y;
     {x:U,y:U}
     y = T:credit_card_no();
     x = T:y;
     {x:T,y:T}
     use(x);
     x = U:0;  // Dead Store
     {x:U,y:T}
     x = T:last_4_digits(y);  // Dead Store
     {x:T,y:T}
     y = U:0;  // Dead Store
     {x:T,y:U}
     return;
}
```

Post-dominated by the next assignment

sat Case 1

sat Case 3

By removing the 2º dead store, we actually obtain a more secure program

INTUITION

- The 1st dead store is post-dominated by the second one
- The 2nd one is always the final assignment to x in every terminating computation, and x is untainted before it

By Case 1, the algorithm would remove the 1st one and keep the 2nd one

Such a transformation is secure, as the source program and result program are both leaky

But Case 3 would do a better job: it will remove the 2nd dead assignment instead

The resulting program is more secure: x becomes untainted at the final location and no private information can be leaked outside the function via x

Secure DSE algorithm: considerations

The algorithm is sub-optimal, given the hardness results, as it may retain more dead stores than necessary

```
void foo()
{
     int x;
     {x:U}
     x = T:read_password();
     {x:T}
     use(x);
     x = T:read_password(); // Dead Store
     {x:T}
     return;
}
```

Store to x is dead and could be securely removed, but it will not by the procedure

Outline

- Dead store elimination and security
- Difficulties for security validation
- Dataflow information
- Information leakage
- Hoare Logic
- Taint Proof System
- Procedure at work
- Final considerations

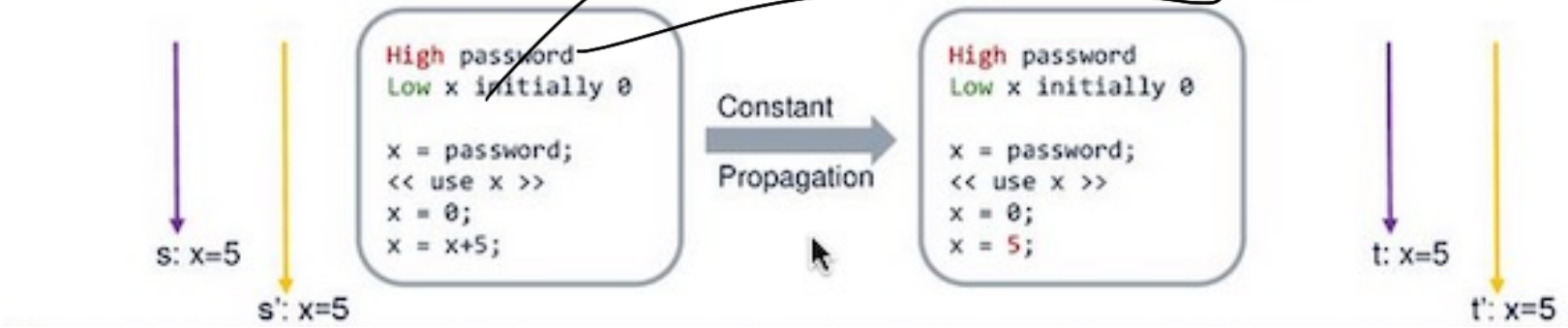
Secure DSE algorithm: considerations

- The algorithm only ensures that no new leaks are added during the transformation, i.e., the transformation is secure
- ~~Correctness is assumed: focus is on information leakage~~

Are other compiler transformations secure?

Theorem: for any transformation with a strict refinement proof, correctness implies security

Several optimizations have strict refinement (DSE does not): e.g., constant propagation, control-flow simplifications, loop unrolling



SSA leaks information

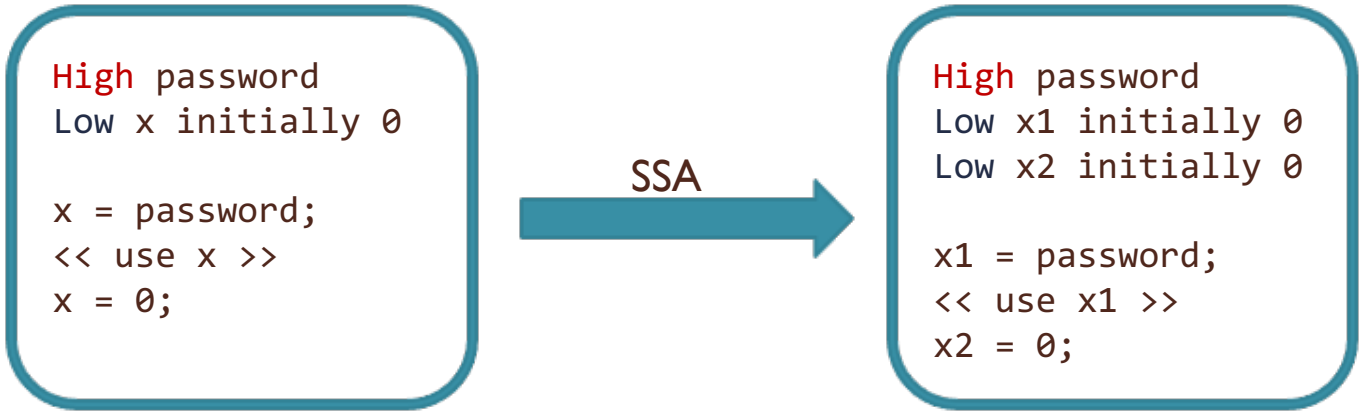
The important single static assignment (SSA) transform is insecure ✓

SSA is a way of structuring the intermediate representation (IR) of programs so that every variable is assigned exactly once and and every variable is defined before it is used

This

- simplifies register allocation by splitting the live range of variables, but
- may expose all intermediate values of variables, which may lead to further leaks

SSA leaks information



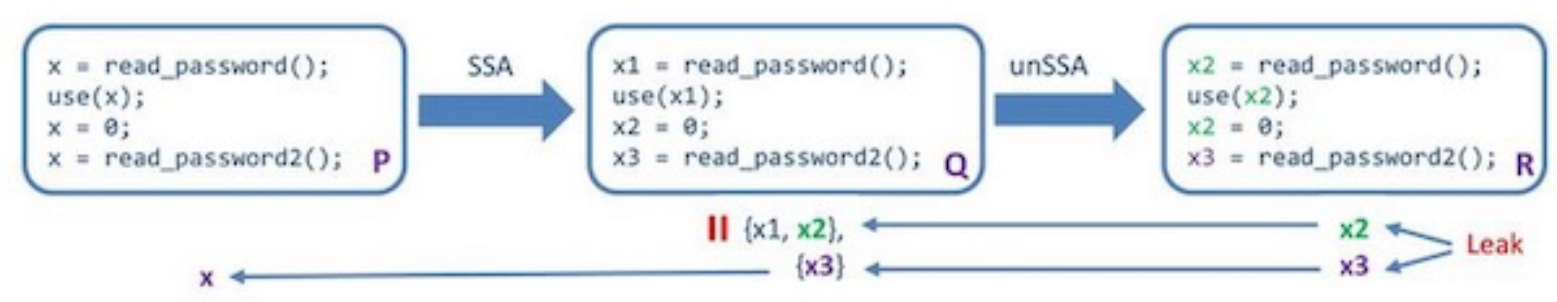
The SSA transform introduces fresh names x1 and x2 for the assignments to x, with different registers. The secret password leaks out through x_1

Possible solutions:

- clear all potentially tainted variables before register allocation: inefficient?
- modify SSA to carry auxiliary information about leakage: how?

SSA is not secure

Sub-optimal grouping using Taint Analysis



Group variants of variables x in Q with mutually disjoint live ranges

Conclusion

- Compiler optimizations may be correct and yet be not secure
- Ensuring security of DSE through translation validation is difficult
- A provably secure DSE transform based on taint propagation + domination

Bibliography

- Chaoqiang Deng, Kedar S. Namjoshi. *Securing a compiler transformation. Formal Methods Syst*. Des. 53(2), 2018. [Journal version]
- Chaoqiang Deng, Kedar S. Namjoshi. *Securing a compiler transformation*. SAS 2016, LNCS, 2016. [Conference version]

End