Language Based Security (LBT)

Secure compilation

Chiara Bodei, Gian-Luigi Ferrari

Lecture May, 10 2024

Outline

Live Variables Analysis
Difficulties for security validation
Syntax and Semantics
Post-domination
Information leakage
Axiomatic semantics
A Taint Proof System
Secure dead store elimination algorithm
Conclusions and what next

Preliminaries: Axiomatic semantics

- Based on formal logic, AS 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

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

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

Hoare triples (cont.)

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

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

Weakest precondition

Weakest precondition (from given postcondition)

“P is stronger than or equal to Q” means “P implies Q”, e.g.,

- "x is a cat" is stronger* than "x is an animal”
- “x > 0” is stronger than “x ≥ 0”

$$P \Rightarrow Q \equiv P \subseteq Q$$

This is an interesting property

There exists also the dual property

Strongest postcondition (from given precondition)

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**

Axiomatic semantics (cont.)

Suppose to have the incomplete triple

{P?} 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

{P?} 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

{P?} 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 weakest 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

Axiomatic program proofs

Example

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

- Which is the procedure to find P??
- 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}

Skip rule

An axiom for Skip

$$\frac{}{\{P\}\ \text{skip}\ \{P\}}$$

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

Assignment rule

An axiom for assignment

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

Syntax replacement

Example, formally:

{P?} 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)
  - P? = Q[E/x] = (a>1)[b+1/a] = b+1 >1
  - Then P? = b>0
- Undo the assignment and solve:

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

Assignment rule

Another example

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

- 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:

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

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

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

Sequential rule

A rule for sequential composition
Second example, formally:

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

$\{P_1?\}$ y = 3*x + 1; x = y + 3 $\{x<10\}$

- $\{P_2?\}$ x = y + 3 $\{x<10\}$
  $\{x<10[y+3/x]\}$ x = y + 3 $\{x<10\}$ $\Rightarrow$ $\{y<7\}$ x = y + 3 $\{x<10\}$
- $\{P_1?\}$ y = 3*x + 1 $\{y<7\}$
  $\{y< 7[3*x+1/y]\}$ y = 3*x + 1 $\{y<7\}$ $\{x<2\}$ $\Rightarrow$ y = 3*x + 1 $\{y<7\}$

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

Inlining

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

$\{x = r + qy\}$

$r := r - y;$

$\{x = r + (q + 1)y\}$

$q := q + 1;$

$\{x = r + qy\}$

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

IF rule

A rule for conditional statement

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

$\{\text{true}\} \equiv$

if $x \geq 0$ do

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

skip;

else

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

$x := -x;$

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

$\{x \geq 0\}$

WHILE rule

There is a rule for the loop statement, but it is quite involving, because the number of iterations cannot always be predetermined

Induction is needed in order to find an invariant

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

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

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

Consequence rule

- Weaken the pre-cond
- Strenghten the post

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

$P \Rightarrow Q \equiv P \subseteq Q$

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

Weakest precondition

$$\{ \qquad\qquad\qquad \} \; x := 3 \; \{ x + y > 0 \}$$

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 $(x + y > 0)$?

$$(y > -3)$$

Partial Correctness

Theorem Any derivable HL triple is sound

Proof: By induction on the derivation tree

Outline

Live Variables Analysis
Difficulties for security validation
Syntax and Semantics
Post-domination
Information leakage
Axiomatic semantics
A Taint Proof System
Secure dead store elimination algorithm
Conclusions and what next

A Taint Proof System

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

Tainted environment

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

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 $\mathcal{E}$

A Taint Proof System (Hoare-style)

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

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

A Taint Proof System (Hoare-style)

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

S *is* skip: $\quad \{\mathcal{E}\}\, \text{skip}\, \{\mathcal{E}\}$

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

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

Taint Proof System: assignment rule

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

x inherits the taint label of e

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

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

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

Assignment rule: examples

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

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();

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

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

c tainted

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

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

Conditional rule: case B

c tainted

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

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

A Taint Proof System: soundness

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 $(s', t') \models \mathcal{F}$.*

Proof:

0) S is skip or $\mathsf{out}(e)$:

$$\{\mathcal{E}\}\, \mathsf{skip}\, \{\mathcal{E}\} \text{ and } \{\mathcal{E}\}\, \mathsf{out}(e)\, \{\mathcal{E}\}$$

Consider states $s = (m, p)$, $t = (n, q)$, $s' = (m', p')$ and $t' = (n', q')$ such that $s \xrightarrow{S} s'$ and $t \xrightarrow{S} t'$ hold. By the semantics of skip and $\mathsf{out}(e)$, $s' = s$ and $t' = t$. Thus, if $(s, t) \models \mathcal{E}$, then $(s', t') \models \mathcal{E}$.

1) S is an assignment $x := e$:

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

A Taint Proof System: soundness

Consider states $s = (m, p)$, $t = (n, q)$, $s' = (m', p')$ and $t' = (n', q')$ such that $s \xrightarrow{S} s'$ and $t \xrightarrow{S} t'$ hold. By the semantics of assignment, it is clear that $p' = p[x \leftarrow p(e)]$, $q' = q[x \leftarrow q(e)]$, and $m' = n'$ denotes the program location immediately after the assignment. Assume $(s, t) \models \mathcal{E}$, we want to prove $(s', t') \models \mathcal{F}$, or more precisely, $\forall v : \neg\mathcal{F}(v) \Rightarrow p'(v) = q'(v)$.

Consider variable y different from x. If $\mathcal{F}(y)$ is false, so is $\mathcal{E}(y)$, hence $p(y) = q(y)$ since $(s, t) \models \mathcal{E}$. As $p'(y) = p(y)$ and $q'(y) = q(y)$, we get $p'(y) = q'(y)$ as desired.

Consider variable x. If $\mathcal{F}(x)$ is false, so is $\mathcal{E}(e)$, hence only untainted variables in $\mathcal{E}$ appear in e. As $(s, t) \models \mathcal{E}$, those variables must have equal values in s and t, thus $p(e) = q(e)$. Since $p' = p[x \leftarrow p(e)]$, $q' = q[x \leftarrow q(e)]$, we know $p'(x) = q'(x)$.

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

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;

  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;
}
```

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

Outline

Live Variables Analysis
Difficulties for security validation
Syntax and Semantics
Post-domination
Information leakage
Axiomatic semantics
A Taint Proof System
Secure dead store elimination algorithm
Conclusions and what next

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

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:

1. the store is post-dominated by other stores to
   - Justification: 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
   - Justification: 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
   - Justification: 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 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

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

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

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

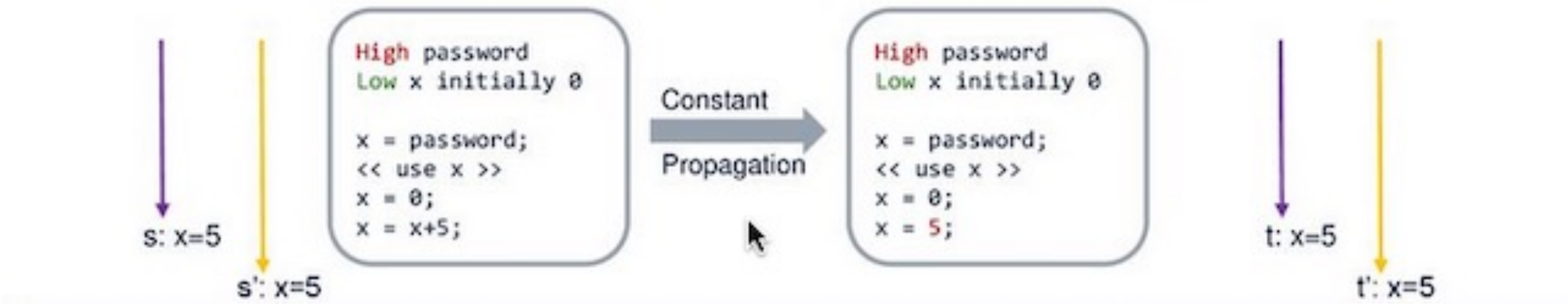
Outline

Live Variables Analysis
Difficulties for security validation
Syntax and Semantics
Post-domination
Information leakage
Axiomatic semantics
A Taint Proof System
Secure dead store elimination algorithm
Conclusions and what next

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

```
High password
Low x initially 0

x = password;
<< use x >>
x = 0;
```

SSA →

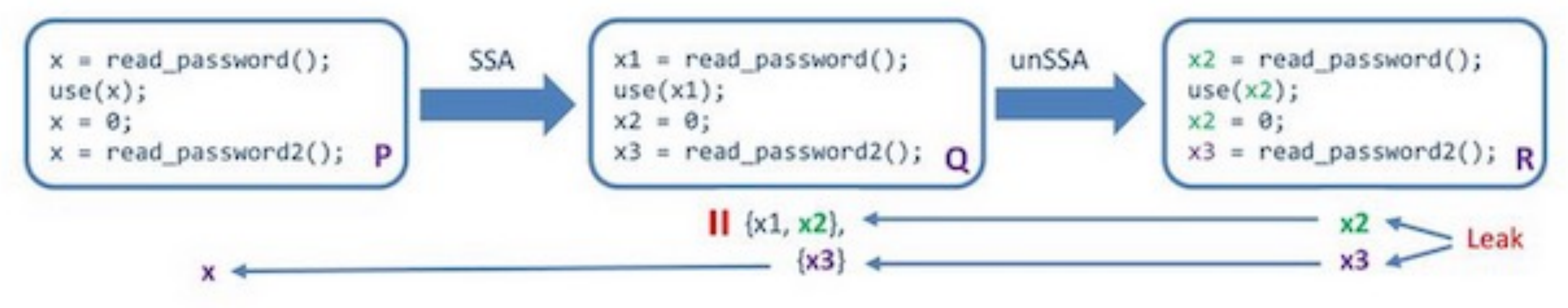
```
High password
Low x1 initially 0
Low x2 initially 0

x1 = password;
<< use x1 >>
x2 = 0;
```

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?

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

- Anders Møller & Michael I. Schwartzbach. *Static Program Analysis* https://cs.au.dk/~amoeller/spa/
- 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