Analysis of security properties

MPRI — Cours 2.6 "Interprétation abstraite : application à la vérification et à l'analyse statique"

Xavier Rival

INRIA

Feb, 12nd, 2021

Security

What does "**security**" mean ?

There are many examples of "**potential security issues**":

- **Leakage of ~~sensible~~ information**: sensitive
  an unauthorized user is able to retrieve or even just guess critical information
- **Code injection**:
  a user succeeds in getting malicious code executed with a high privilege (and can corrupt data or take control of the system)
- **Authentication breach**:
  a malicious user pretends to be another user

"Security" is a rather general and vague term.
We need to be more specific on what it means.

Rough intuition:

- **safety issue**, *e.g.*, runtime error: failure in presence of just the environment
- **security issue**: failure resulting from (malicious) **deliberate user action**

Objectives of this course

1. **Understand the difficulty inherent in security properties**:
   In general, security properties are significantly harder to reason about than safety properties

2. **Introduce hyperproperties**:
   A **more general framework** than the trace properties we are used to, which can express many relevant program properties

3. **Describe a few abstractions for security**:
   - extension of abstractions for safety
   - specific abstractions

In one class, we can only provide an introduction to the field.
Our goal is to understand the main problems.

Outline

leakage

Notations

We focus on imperative programs viewed as **transition systems**:

- set of **control states**: $\mathbb{L}$ (program points)
- set of **variables**: $\mathbb{X}$ (all assumed globals)
- set of **values**: $\mathbb{V}$
- set of **memory states**: $\mathbb{M}$ $= \mathbb{X} \to \mathbb{V}$
- set of **states**: $\mathbb{S} = \mathbb{L} \times \mathbb{M}$ and **initial states** $\mathbb{S}_\mathrm{i} \subseteq \mathbb{S}$
- **transition relation**: $(\rightarrow) \subseteq \mathbb{S} \times \mathbb{S}$, assumed $\underbrace{\text{deterministic}}_{\text{bold}}$

Semantics:

- **reachable states**: $[\![P]\!]_\mathcal{R} \subseteq \mathbb{S}$ $\qquad [\![P]\!]_\mathcal{R} = \mathrm{lfp}_\emptyset F_\mathcal{R}$
- **finite execution traces**: $[\![P]\!]_{\mathcal{T}^{*\omega}} \subseteq \mathbb{S}^*$
- **denotational semantics**: $[\![P]\!]_\mathcal{F} : \mathbb{S} \longrightarrow \mathcal{P}(\mathbb{S})$
  where $[\![P]\!]_\mathcal{F}(s) = \{s' \in \mathbb{S} \mid s \rightarrow^* s'\}$

  Given $\ell, \ell' \in \mathbb{L}$, we also let $[\![P]\!]_{\mathcal{F}[\ell,\ell']} : \mathbb{M} \longrightarrow \mathcal{P}(\mathbb{M})$ be defined by
  $[\![P]\!]_{\mathcal{F}[\ell,\ell']}(m) = \{m' \in \mathbb{M} \mid (\ell, m) \rightarrow^* (\ell', m')\}$.

For us today, most of the time we use the **denotational semantics**.

Non-interference

Among the many possible security properties, we choose one, that is very representative.

It describes the fact that **some secret information should not be guessed (directly or indirectly) by any unauthorized user.**

Non-interference (informal definition)

Notations:

- $\mathbb{X}_{\text{pub}}$: **public variables**, observed by anybody (also called "low", *i.e.*, it requires only a low authorization)
- $\mathbb{X}_{\text{sec}}$: **secrete variables**, should not be observed by anybody, save authorized users (also called "high", *i.e.*, high authorization)

We say that a program P satisfies the **non-interference** property defined by $\mathbb{X}_{\text{pub}}, \mathbb{X}_{\text{sec}}$ if and only if any execution of the program where one can only observe the values of the variables in $\mathbb{X}_{\text{pub}}$ does not allow to derive any information about the values of the variables in $\mathbb{X}_{\text{sec}}$.

This definition is quite informal, and we will make it precise and formal later.

Example of program violating non-interference

Let us consider the program below:

```
int s; // private variable, should be secure
int i; // public variable, can be seen by anybody

s = private_computation( ); // should remain secret

i = s + 8;
// anyone can observe i here!
```

s = 4992

i = 5000

We should let:

- $\mathbb{X}_{\text{pub}} = \{\mathtt{i}\}$
- $\mathbb{X}_{\text{sec}} = \{\mathtt{s}\}$ (for readability we will write s for the private variable that should remain secure)

This program **clearly violates non-interference**.

If we know the final value of i we can subtract 8 and derive the value of s

Example of program satisfying non-interference

We now consider the program below, with the same $\mathbb{X}_{\mathrm{pub}}, \mathbb{X}_{\mathrm{sec}}$:

```
int s; // private variable, should be secure
int i; // public variable, can be seen by anybody

s = private_computation( ); // should remain secret

i = user_input( ) + 8;
// anyone can observe i here!
```

– crash
– not terminate
ignore

no leak
i = 5000

This program **satisfies non-interference**.

The final value of i **is computed in a way that is never influenced by that of** s (the user input ignores the value of s at this point).

A few more subtle cases

We use the **same conventions** (with variables i, s).

Program 1: non-interference **violated**

==

```
// ... as before, s stores the secret
if( s = 7 )
  i = 1;
else
  i = -1;
```

s

i := β(s)

expr

There is an **implicit information flow**. If we observe that i is 1, we know exactly what s is.

Program 2: non-interference **violated**

```
s = 8 / s;
i = 5;
```

Again, there is an **implicit information flow**. If we observe a crash (no value for i), we know that $s = 0$.

Program 3: non-interference **satisfied**

i

```
x = 0 * s;
```

There is **no information flow**. Indeed, i is 0 regardless...

In the following, we need to **formalize and characterize non-interference** before we can actually reason about it.

Non-interference

A couple of **caveats**:

- **termination** may change observation (though we cannot positively observe non-termination)
- **errors** may change observation too

For the sake of simplicity, we **ignore** these and consider **termination insensitive non-interference** and do not consider errors. In the following, we still call this notion **non-interference**.

Observation point: we search whether **public variables observed at end point** $\ell_\dashv$ reveal anything about **private variables observed at entry point** $\ell_\vdash$.

$(l_\vdash, m) \longrightarrow^* (l_\dashv, m')$

Non-interference (formal definition)

We say that a program P satisfies the **non-interference** property defined by $\mathbb{X}_{\mathrm{pub}}/\ell_\dashv, \mathbb{X}_{\mathrm{sec}}/\ell_\vdash$ if and only if for all memory states $m_0, m_1 \in \mathbb{M}$,

$$(\forall \mathrm{x} \in \mathbb{X}_{\mathrm{pub}},\ m_0(\mathrm{x}) = m_1(\mathrm{x})) \\ \Longrightarrow (\forall \mathrm{x} \in \mathbb{X}_{\mathrm{pub}},\ [\![P]\!]_{\mathcal{F}[\ell_\vdash,\ell_\dashv]}(m_0)(\mathrm{x}) = [\![P]\!]_{\mathcal{F}[\ell_\vdash,\ell_\dashv]}(m_1)(\mathrm{x}))$$

Outline

Semantics

$[\![P]\!]_{\mathcal{T}^*} \subseteq \mathbb{S}^*$
$\subseteq (\mathbb{S} \times \mathbb{N})^*$
time information

We have seen three semantics, **that are comparable**:

- **trace semantics** is the most precise and informative
  i.e., the denotational semantics can be computed from it
- then **denotational semantics** is more precise than reachable states
  i.e., the reachable states semantics can be computed from it

> Before we formalize and study non-interference,
> we recall a few important points about trace properties.

To study hierarchies of properties, the most expressive semantics is more adapted.

Recall:

- **finite traces semantics** $[\![P]\!]_{\mathcal{T}^*} \subseteq \mathbb{S}^*$
  expressed as a least fixpoint
- **infinite traces semantics** $[\![P]\!]_{\mathcal{T}^\omega} \subseteq \mathbb{S}^\omega$
  expressed as a greatest fixpoint
- **all traces semantics** $[\![P]\!]_{\mathcal{T}^{*\omega}} \subseteq \mathbb{S}^{*\omega} = \mathbb{S}^* \uplus \mathbb{S}^\omega$
  $([\![P]\!]_{\mathcal{T}^{*\omega}} = [\![P]\!]_{\mathcal{T}^*} \uplus [\![P]\!]_{\mathcal{T}^\omega})$
  can also be expressed as a fixpoint, by fixpoint combination technique

Semantic properties as sets of behaviors

We consider the following model:

- a semantic property is describes by the set of behaviors that are compliant with
- a program satisfies such a property if and only if all the behaviors of the program are compliant with the property, *i.e.*, are elements of the set describing the property

Direct formalization:

Definition: semantic property (or verification goal)

Assuming **program behaviors** range in set $\mathcal{S}$, a **semantic property** is a set $G \subseteq \mathcal{S}$.

↳ traces

Given program P, then P satisfies G if and only if:

$$\forall b \in [\![P]\!]_{\mathcal{S}},\ b \in G$$

or equivalently,

$$[\![P]\!]_{\mathcal{S}} \subseteq G$$

$$\boxed{[\![P]\!]_{\tau^{*\omega}} \subseteq G}$$

Trace property

Let us see some examples...

Examples

Unreachability of certain states $S \subseteq \mathbb{S}$:

- set property $\mathbb{S} \setminus S$
  i.e., we want to show $[\![P]\!]_{\mathcal{R}} \subseteq \mathbb{S} \setminus S$
- trace property $(\mathbb{S} \setminus S)^{*\omega}$
  i.e., we want to show $[\![P]\!]_{\mathcal{T}^{*\omega}} \subseteq (\mathbb{S} \setminus S)^{*\omega}$
- classical case 1: S corresponds to error states or dangerous states (**absence of runtime errors**)
- classical case 2: S corresponds to exit state that violate some exit condition (**partial correctness**)

Termination:

- trace property $(\mathbb{S})^*$
  i.e., we want to show $[\![P]\!]_{\mathcal{T}^{*\omega}} \subseteq (\mathbb{S})^*$
  or, equivalently that $[\![P]\!]_{\mathcal{T}^{\omega}} \subseteq \emptyset$ (*i.e.*, P has no infinite trace)

Depending on property kinds, specific **proof methods/analysis methods** apply...

Outline

Safety properties

Informal definition:

A **safety property** is a semantic property such that,
when it does not hold, it admits a **finite counter-example trace**

Intuitively, we can **test** a safety property on a trace and be able to say after a finite session whether this trace is a counter-example or not.

Definition: safety property

A trace property $S \subseteq \mathbb{S}^{*\omega}$ is a **safety property** if and only if:

$$\forall T \subseteq \mathbb{S}^{*\omega},\ T \not\subseteq S \Longrightarrow \exists \sigma \in \mathbb{S}^*,\ \exists \sigma' \in \mathbb{S}^{*\omega},\ \sigma \cdot \sigma' \in T,\ \wedge \forall \sigma'' \in \mathbb{S}^*,\ \sigma \cdot \sigma'' \notin S$$

If we let $T = [\![P]\!]_{\mathcal{T}^{*\omega}}$, we recover the informal definition.

Remarks:

- infinite traces do not matter, thus we can consider $[\![P]\!]_{\mathcal{T}^*}$ instead of $[\![P]\!]_{\mathcal{T}^{*\omega}}$;
- if we could enumerate all finite traces of P we can decide whether it satisfies safety property S.

Examples of safety properties

State properties are safety properties:

- let us consider $G \subseteq \mathbb{S}$ and state property G
- then if we consider traces,

$$[\![P]\!]_{\mathcal{R}} \subseteq G \iff [\![P]\!]_{\mathcal{T}^*} \subseteq G^* \quad \text{which is safety}$$

  proof left as exercise
- **consequence**:
  absence of runtime errors and functional correctness are safety

But **many interesting safety properties are *not* state properties**:

- let $s \in \mathbb{S}$
- consider S defined by $\langle s_0, \ldots, s_n \rangle \in S \iff \mathbf{Card}(\{i \in \mathbb{N} \mid s_i = s\}) \leq 1$
  i.e., a trace is correct if and only if it cannot visit s twice
- we can show that S is a safety property
- based on the states it visits, one cannot say whether a trace meets it

Proof method for safety

We consider a program P with initial states $\mathbb{S}_\mathrm{i}$ and transition relation $\rightarrow$:

Principle of invariance proofs

Let $\mathbb{I}$ be a set of finite traces; it is said to be an **invariant** if and only if:

- $\forall s \in \mathbb{S}_\mathcal{I}, \langle s \rangle \in \mathbb{I}$
- $\forall \langle s_0, \ldots, s_n \rangle \in \mathbb{I}, \forall s_{n+1} \in \mathbb{S}, s_n \rightarrow s_{n+1} \Longrightarrow \langle s_0, \ldots, s_{n+1} \rangle \in \mathbb{I}$

It is stronger than S if and only if $\mathbb{I} \subseteq S$.

The **"by invariance"** proof method is based on finding an invariant that is stronger than $\mathcal{T}$.

This proof method always works (theorem proof left as an exercise):

Theorem: soundness and completeness

S safety
P

$[\![P]\!] \subseteq S \Longleftrightarrow \exists \mathcal{I}$ inv.
$\mathcal{I} \subseteq S$

A safety property holds if and only if there exists an invariant stronger than it

But, finding a suitable invariant $\mathbb{I}$ is often **very difficult** (especially automatically)

Liveness properties

Informal definition, a form of dual of safety:

A **liveness property** is a semantic property such that any finite execution may be extended into a correct one; thus, it has **no finite counter-example**

Canonical example: **termination**

- after finitely many steps of an unfinished execution, we cannot say for sure whether the program **is about to terminate** or **will never terminate**...
- **consequence**: testing will **not** produce counterexample for termination
  hack: search for repeating states in finite executions
  but this is changing the problem and will not capture all cases of NT

Definition: liveness property

A trace property $L \subseteq \mathbb{S}^{*\omega}$ is a **liveness property** if and only if:

$$\forall \sigma \in \mathbb{S}^*,\ \exists \sigma' \in \mathbb{S}^{*\omega},\ \sigma \cdot \sigma' \in L$$

Termination:

$L = \mathbb{S}^*$, *i.e.*, there should be no infinite trace

Proof method for liveness

There exists also a **proof method for liveness properties**, which is also sound and complete.

We only sketch the **case of termination** since the general proof principle is long to describe and similar in spirit...

Definition: ranking function

A **ranking function** for program P is a function $\phi : \mathbb{S}^* \longrightarrow E$, where E with partial order $\preceq$ is a **well-founded ordering** (no infinite decreasing chains) and the **ranking property** below holds:

$$\begin{array}{l} \forall \langle s_0, \ldots, s_n \rangle,\ \forall s_{n+1} \in \mathbb{S}, \\ \qquad s_n \rightarrow s_{n+1} \Longrightarrow \phi(\langle s_0, \ldots, s_{n+1} \rangle) \prec \phi(\langle s_0, \ldots, s_n \rangle) \end{array}$$

This is the basis for proof methods that reduce the **search of a variant** (like a ranking function) to that of an **invariant**, but **for a different program**.

Decomposition of trace properties

Theorem: decomposition (Alpern & Schneider 88)

Let $T \subseteq \mathbb{S}^{*\omega}$; it can be decomposed into the **conjunction** of a **safety property** S and a **liveness property** L:

$$T = S \cap L$$

Proof:

- it is actually **systematic** and **constructive**
  i.e., it describes precisely how both S and L can be defined
- see the paper for details (part of recommended reading assignment)

Application: **how to verify any trace property T**

1. **decompose** it into $T = S \cap L$ where S is a safety property and L a liveness property
2. **search for an invariant** to prove S
3. **search for a variant** to prove L

difficult

Example: total correctness
S: absence of crashes + partial correctness and L: termination

Status so far

$T = S \wedge L$

T_{NI}

Trace properties
total correctness

Safety properties
never reach s_0 before s_1

$\mathring{S}$

State properties
absence or runtime errors
partial correctness

PM

Liveness properties
termination

$\mathring{L}$

PM

- actually there is a small interaction between safety and liveness
- proof methods exist for all these
- we can search for invariants by static analysis...

Outline

Refinement: monotonicity over behaviors and properties

Monotonicity over properties

T_0 stronger than T_1

Let T_0, T_1 be two trace properties such that $T_0 \subseteq T_1$.
Let P be a program. Then:

$[\![P]\!] \subseteq T_0 \wedge T_0 \subseteq T_1 \Longrightarrow [\![P]\!] \subseteq T_1$

If P satisfies T_0, then P satisfies T_1.

- obvious consequence of the definition using $\subseteq$
- intuitively, a property that consist of fewer behaviors is **stronger**

Monotonicity over program behaviors

P_0 : more behaviors than P_1

Let P_0, P_1 be two programs such that $[\![P_0]\!]_{\mathcal{T}^{*\omega}} \supseteq [\![P_1]\!]_{\mathcal{T}^{*\omega}}$.
Let T be a trace property. Then:

If P_0 satisfies T, then P_1 satisfies T.

- again, obvious consequence of the definition using $\subseteq$
- intuitively, a program with fewer behaviors **satisfies more properties**.

Monotonicity over program behaviors also holds if we consider $[\![.]\!]_{\mathcal{R}}$ or $[\![.]\!]_{\mathcal{F}[.,.]}$ instead of $[\![.]\!]_{\mathcal{T}^{*\omega}}$

Two (contrived) examples programs and non-interference

A few **simplifying assumptions** (it is hard to do simpler...):

- only two variables $\mathtt{s}, \mathtt{x}$, with $\mathtt{s}$ private and $\mathtt{x}$ public
  thus $\mathbb{X}_{\mathrm{pub}} = \{\mathtt{x}\}$ and $\mathbb{X}_{\mathrm{sec}} = \{\mathtt{s}\}$
- only two values $\mathbb{V} = \{0, 1\}$
- for clarity we write $(m(\mathtt{x}), m(\mathtt{s}))$ for the memory state m

$\rightarrow (m(s), m(x))$

We consider P_0, P_1 with the denotational semantics below

$$\llbracket P_0 \rrbracket_{\mathcal{F}[\ell_\vdash, \ell_\dashv]} : \begin{array}{ccc} (0,0) & \longmapsto & \mathbb{M} \\ (0,1) & \longmapsto & \mathbb{M} \\ (1,0) & \longmapsto & \mathbb{M} \\ (1,1) & \longmapsto & \mathbb{M} \end{array} \quad \supseteq \quad \llbracket P_1 \rrbracket_{\mathcal{F}[\ell_\vdash, \ell_\dashv]} : \begin{array}{ccc} (0,0) & \longmapsto & \mathbb{M} \\ (0,1) & \longmapsto & \mathbb{M} \\ (1,0) & \longmapsto & \{(1,1)\} \\ (1,1) & \longmapsto & \{(1,1)\} \end{array}$$

✓ ×

$P_1 \longrightarrow (?, 0) \rightarrow$ pub

Observations:

- P_0 **satisfies non-interference**:
  whatever the private input, the public output is always 1, thus there is no way to learn anything about the secret
- P_1 **violates non-interference**:
  when the public output is 0, we know the private input **cannot be** 1

×

Non interference is not a trace property

Let us put it all together:

- P_0 **has more behaviors than** P_1
- P_0 **satisfies non-interference**
- thus, **if non-interference was a trace property** then P_1 **should satisfy non-interference**
- **but** P_1 **violates non-interference**

Conclusion:

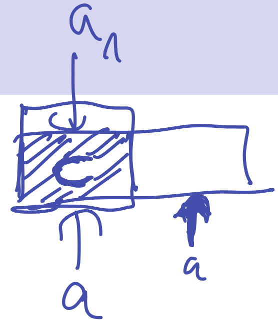
Non-interference is not a trace property.

i.e., we cannot characterize non-interference by a set of "non-interfering" executions...

Consequences:

- we cannot decompose it into safety/liveness and apply existing proof methods, and apply directly previously shown static analysis methods
- we **need to study different techniques**

Outline



"Attacker"

A : commands

A ::= ___
| ___
| write(a)
| ε

Absence of code injection

$\forall \; \underline{A_0}, \; \underline{A_1}$

$P | A_0 \qquad\qquad P | A_1$

$(C) \equiv (C)$

a_1

$\neq$

Moving to sets of sets of behaviors

We first search for **how to characterize** non-interference (and related security properties):

Definition: semantic hyperproperty

Assuming **program behaviors** range in set $\mathcal{S}$ a **semantic hyperproperty**, a semantic property is a set of sets $\mathcal{G} \subseteq \mathcal{P}(\mathcal{S})$.
Given program P, then P satisfies $\mathcal{G}$ if and only if:

$$[\![P]\!]_{\mathcal{S}} \in \mathcal{G}$$

Important differences with everything we have seen so far:

- **all** executions of the program are considered **at once**
  i.e., adding or removing one trace may invalidate the property of the **whole** set
- known proof methods/static analysis techniques **break**
  i.e., we cannot check execution traces one by one (by testing)
  i.e., we cannot rely on an over-approximation of $[\![P]\!]_{\mathcal{S}}$
  (that could be computed by static analysis)

Properties as hyperproperties

Lemma

Any trace property can be described by a semantically equivalent hyperproperty.

Indeed, let $T \subseteq \mathbb{S}^{*\omega}$ be a trace property and P a program. Then:

$$\begin{array}{rcl} P \text{ satisfies } T & \Longleftrightarrow & [\![P]\!]_{\mathcal{T}^{*\omega}} \subseteq T \\ & \Longleftrightarrow & [\![P]\!]_{\mathcal{T}^{*\omega}} \in \mathcal{P}(T) \end{array}$$

Thus **property** T describes the same program as **hyperproperty** $\mathcal{P}(T)$ (powerset induces a downwards closure on hyperproperties).

Note that:

- the monotonicity results **do not hold for hyperproperties**
- for specific pairs of hyperproperties, we may of course observe a monotone behavior, *e.g.* for hyperproperties induced by properties.

Non-interference

To express non-interference on traces we need to **abstract traces into input-output functions**:

$$\begin{array}{rccl} \Phi: & \mathbb{S}^{*\omega} & \longrightarrow & (\mathbb{M} \longrightarrow \mathcal{P}(\mathbb{M})) \\ & T & \longmapsto & \lambda m \cdot \{m' \in \mathbb{M},\ \langle (\ell_{\vdash}, m), \ldots, (\ell_{\dashv}, m') \rangle \in T\} \end{array}$$

We can now define **non-interference** as an **hyperproperty**:

$$\begin{array}{rcl} \mathcal{N} & = & \{T \in \mathcal{P}(\mathbb{S}^{*\omega}) \mid \\ & & \quad \forall m_0, m_1 \in \mathbb{M}, (\forall \mathrm{x} \in \mathbb{X}_{\mathrm{pub}},\ m_0(\mathrm{x}) = m_1(\mathrm{x})) \\ & & \quad\quad \Longrightarrow \forall \mathrm{x} \in \mathbb{X}_{\mathrm{pub}},\ \Phi(T)(m_0)(\mathrm{x}) = \Phi(T)(m_1)(\mathrm{x}) \ \} \end{array}$$

This definition captures the non-interference property:
whenever two initial memories agree on public variables
then corresponding final states should agree on private variables.

Examples (continued):

- $[\![P_0]\!] \in \mathcal{N}$
- $[\![P_1]\!] \notin \mathcal{N}$

Average execution time

We temporarily make a few **limiting assumptions** on programs:

- we consider **only terminating programs**
- we consider **only programs with finitely many complete executions**
  complete executions: from entry control state $\ell_\vdash$ to exit control state $\ell_\dashv$

Given a set of traces $T \in \mathcal{P}(\mathbb{S}^*)$, we define:

$$\text{Avg\_len}(T) = \frac{1}{|T|} \sum_{\sigma \in T} \textbf{length}(\sigma)$$

where **length** returns the length of a trace.

Average execution time lower than $k \in \mathbb{N}$ (clearly not a trace property):

$$\mathcal{A}_k = \{T \in \mathcal{P}(\mathbb{S}^*) \mid \text{Avg\_len}(T) \leq k\}$$

$\hookrightarrow T \ni \sigma \rightarrow |\sigma|$

Generalization:

- with some **measure theory**, we can extend similar properties to **infinite sets of program traces**
- we can also **let programs have some infinite traces**, and consider only the finite ones

Interesting families of hyperproperties

Can we divide the set of hyperproperties in interesting sub-classes ?

Hierarchy inspired by the **safety/liveness** division,
and more precisely **how can a hyperproperty be disproved**:

- **hypersafety**:
  can always be disproved using a **finite** set of **finite** traces
- *k*-**safety**:
  can always be disproved using a set of **at most k finite** traces
  clearly:
  - k-safety hyperproperties are also $k+1$-safety
  - k-safety hyperproperties are also hypersafety
- **hyperliveness**:
  disproving them requires looking at **infinite traces** or **infinite sets of traces**

We now formalize some of these sets more in detail...

Hypersafety

The idea is to extend safety, except that the observation is limited to finite sets finite traces, instead of just finite traces.

Extension of an observation:
Given $T, T' \subseteq \mathbb{S}^{*\omega}$, we say that T' extends T and note $T \leq T'$ if and only if:

$$\forall \sigma \in T,\ \exists \sigma' \in \mathbb{S}^{*\omega},\ \sigma \cdot \sigma' \in T'$$

Definition: hypersafety

Let $\mathcal{G} \in \mathcal{P}(\mathcal{P}(\mathbb{S}^{*\omega}))$ be a hyperproperty. Then, we say that $\mathcal{G}$ is a **hypersafety** property if and only if for all $T \in \mathcal{P}(\mathbb{S}^{*\omega})$, if T does not satisfy $\mathcal{G}$, then

$$\exists M \subseteq \mathbb{S}^{*\cancel{\omega}},\ \begin{cases} & M \text{ is a finite set} \\ \wedge & M \leq T \\ \wedge & \forall T' \subseteq \mathbb{S}^{*\omega},\ M \leq T' \Longrightarrow \cancel{M} \notin \mathcal{G} \end{cases}$$

Faute!

T'

Examples:

- absence of runtime errors (counter-example: one crashing trace)
- non-interference (counter-example: two traces revealing leak)

k-safety

Hypersafety is **not very specific**, as counter-examples can be **arbitrarily large**.

Additional (parametric) restriction: the number of traces in the counter-example.

Definition: k-safety

Let $\mathcal{G} \in \mathcal{P}(\mathcal{P}(\mathbb{S}))$ be a hyperproperty. Then, we say that $\mathcal{G}$ is a ***k*-safety** property if and only if for all $T \in \mathcal{P}(\mathbb{S}^{*\omega})$, if T does not satisfy $\mathcal{G}$, then

$$\exists M \subseteq \mathbb{S}^{*\omega}, \left\{ \begin{array}{ll} & M \text{ has at most } k \text{ elements} \\ \wedge & M \leq T \\ \wedge & \forall T' \subseteq \mathbb{S}^{*\omega},\ M \leq T' \Longrightarrow M \notin \mathcal{G} \end{array} \right.$$

Interesting examples:

↳ 2-safety

- **all safety properties** are **1-safety**
  i.e., counter-examples consist only of one offending finite trace this includes the absence of runtime errors
- **non-interference**:
  i.e., by the definition a counter-example is made of two finite traces

Hyperliveness

Intuition behind liveness: finite observations are not counter-examples.

We can extend this intuition here, except that a finite observation is now any finite set of finite execution traces:

Definition: hyperliveness

Let $\mathcal{G} \in \mathcal{P}(\mathcal{P}(\mathbb{S}))$ be a hyperproperty. Then, we say that $\mathcal{G}$ is a **hyperliveness** property if and only if

$$\forall T \subseteq \mathcal{P}(\mathbb{S}^{*\omega}),\ T \text{ finite} \Longrightarrow \exists T' \subseteq \mathcal{P}(\mathbb{S}^{*\omega}),\ \left\{ \begin{array}{rl} & T \leq T' \\ \wedge & T' \in \mathcal{G} \end{array} \right.$$

Example:

the average run-time is less than N steps

Indeed, any finite set of executions may be extended with enough short ones to bring down the average.

Decomposition of hyperproperties

biblio

We can also extend the **Alpern & Schneider decomposition theorem**:

Decomposition theorem

Let $\mathcal{G} \in \mathcal{P}(\mathcal{P}(\mathbb{S}^{*\omega}))$ be a hyperproperty. Then, there exist

- a hypersafety property $\mathcal{S}$ and
- a hyperliveness property $\mathcal{L}$

such that:

$$\mathcal{G} = \mathcal{S} \cap \mathcal{L}$$

In the following of this class, though **2-safety is enough**.

Outline

From traces to sets of traces in the semantics

Observations so far:

- typical semantics describe **sets of behaviors** and are based on **fixpoint definitions**
- abstract interpretation builds upon abstraction and fixpoint definition hence, it allows to over-approximate **sets of behaviors**
- in the case of non-interference, **over-approximating sets of behaviors** is **not useful**
  the same goes for any hyperproperty that is not a trace property...

We need a technique to conservatively reason over hyperproperties

We are going to consider **two approaches**:

1. **lifting the semantics** to sets of sets of traces
2. **re-expressing** the hyperproperties that we are interested in

A very basic language

In the following, we study a very basic imperative language, to describe a static analysis based on a semantics defined in terms of sets of sets:

- as before, we assume finitely many variables $\mathbb{X}$ and a set set of base type values $\mathbb{V}$
- **expressions**:

$$\begin{array}{rcll} \mathtt{e} & ::= & \mathtt{v} & \text{base type value} \\ & | & \mathtt{x} & \text{variable} \\ & | & \mathtt{e}_0 \oplus \mathtt{e}_1 & \text{binary operation } \oplus \end{array}$$

- **commands**:

$$\begin{array}{rcll} \mathtt{s} & ::= & \mathtt{x} := \mathtt{e} & \text{assignment} \\ & | & \mathbf{skip} & \text{do nothing} \\ & | & \mathtt{s}_0; \mathtt{s}_1 & \text{sequence} \\ & | & \mathbf{if}(\mathtt{e}_0)\ \mathtt{s}_1\ \mathbf{else}\ \mathtt{s}_2 & \text{condition} \\ & | & \mathbf{while}(\mathtt{e}_0)\ \mathtt{s}_1 & \text{loop} \end{array}$$

- non-determinism occurs **only at the beginning of program execution**
  once the initial state is set up, no non-determinism occurs

Base semantics: function over sets of relations

We are interested in **input-output** relations:

- **standard approach**: map input memory state into output memory state
- to obtain **more general statements**: functions over such pairs
  1. $[\![P]\!]_{\mathrm{rel}}$ inputs (m_0, m_1), assumes that a previous run from m_0 led to m_1
  2. it computes the effect of P from there, we assume the result is m_2
  3. then, it returns the new pair $(m_0, m_2) \in \mathbb{F} = \mathbb{M} \times \mathbb{M}$

Semantics of expressions ($[\![\mathrm{e}]\!] : \mathbb{M} \longrightarrow \mathbb{V}$):

$$[\![\mathrm{v}]\!](m) = \mathrm{v} \qquad [\![\mathrm{x}]\!](m) = m(\mathrm{x}) \qquad [\![\mathrm{e}_0 \oplus \mathrm{e}_1]\!](m) = [\![\mathrm{e}_0]\!](m) \bar{\oplus} [\![\mathrm{e}_1]\!](m)$$

Semantics of commands ($[\![\mathrm{s}]\!] : \mathbb{F} \uplus \{\bot\} \longrightarrow \mathbb{F} \uplus \{\bot\}$):

$$\begin{array}{rcl}
[\![\mathrm{s}]\!](\bot) & = & \bot \\
[\![\mathrm{x} := \mathrm{e}]\!](m_0, m_1) & = & (m_0, m_1[\mathrm{x} \mapsto [\![\mathrm{e}]\!](m_1)]) \\
[\![\mathbf{skip}]\!](m_0, m_1) & = & (m_0, m_1) \\
[\![\mathrm{s}_0; \mathrm{s}_1]\!](m_0, m_1) & = & [\![\mathrm{s}_1]\!] \circ [\![\mathrm{s}_0]\!](m_0, m_1) \\
[\![\mathbf{if}(\mathrm{e}_0)\ \mathrm{s}_1\ \mathbf{else}\ \mathrm{s}_2]\!](m_0, m_1) & = & \left\{ \begin{array}{ll} [\![\mathrm{s}_1]\!](m_0, m_1) & \text{if } [\![\mathrm{e}]\!](m_1) = \mathrm{true} \\ [\![\mathrm{s}_2]\!](m_0, m_1) & \text{if } [\![\mathrm{e}]\!](m_1) = \mathrm{false} \end{array} \right. \\
[\![\mathbf{while}(\mathrm{e}_0)\ \mathrm{s}_1]\!](m_0, m_1) & = & \mathbf{lfp}\ G \\
 & & \text{where } G \text{ is left as an exercise}
\end{array}$$

pre
-continuat

$$\begin{tikzcd} m_0 \arrow[r] & m_1 \arrow[r, "S"] \arrow[r, bend right, "{[s]}"'] & m_2 \end{tikzcd}$$

(m_0, m_1) $\quad$ (m_0, m_2)

$\cup$

ε

(m, m)

Non-interference

We can express **non-interference** directly.
Assumption: $\mathbb{X}_{\mathrm{pub}}, \mathbb{X}_{\mathrm{sec}}$ are given.

We let the following equivalence relation describe **memory agreement** on any given set of variables X:

- notation: $m_0 \equiv_X m_1$
- condition:

$$m_0 \equiv_X m_1 \iff \forall \mathrm{x} \in X,\ m_0(\mathrm{x}) = m_1(\mathrm{x})$$

Non-interference (normal) semantics level

Program P satisfies non-interference if and only if

$$\forall m_0, m_0', m_1, m_1' \in \mathbb{M},\quad \left.\begin{array}{l} m_0 \equiv_{\mathbb{X}_{\mathrm{pub}}} m_0' \\ \wedge \quad [\![P]\!](m_0) = (m_1) \\ \wedge \quad [\![P]\!](m_0') = (m_1') \end{array}\right\} \Longrightarrow m_1 \equiv_{\mathbb{X}_{\mathrm{pub}}} m_1'$$

$[\![P]\!](m_0, m_1) = (m_0, m_2)$

$(m_0', m_1') = (m_0', m_2')$

$m_2 \equiv_{\mathbb{X}_{\mathrm{pub}}} m_2'$

Remark: we could work out similar definitions with full traces rather than relations...

Towards a non-standard semantics

Base semantics:

- we have defined $[\![\mathrm{s}]\!] : \mathbb{F} \uplus \{\perp\} \longrightarrow \mathbb{F} \uplus \{\perp\}$
- let $\delta_M = \{(m, m) \mid m \in M\}$
- then, $[\![\mathrm{s}]\!](\delta_{\mathbb{M}})$ describes exactly the input/output pairs of s
  as observed, over-approximating this set of pairs is of **no use** to prove non-interference, thus we turn to **a new semantics**

Hypercollecting semantics:

- goal: compute a set of set of pairs...
- thus, we let $[\![\mathrm{s}]\!]_{\mathcal{H}} : \mathcal{P}(\mathcal{P}(\mathbb{F})) \longrightarrow \mathcal{P}(\mathcal{P}(\mathbb{F}))$ and $\Delta_{\mathbb{M}} = \{\delta_M \mid M \in \mathcal{P}(\mathbb{M})\}$
- then, $[\![\mathrm{s}]\!]_{\mathcal{H}}(\Delta_{\mathbb{M}})$ computes the set of sets of input/output pairs, for any set of inputs

We will set up the definition of $[\![.]\!]_{\mathcal{H}}$ so as to meet the following two conditions:

1. for all s and for all $F \in \mathcal{P}(\mathbb{F})$, the definition of $[\![\mathrm{s}]\!]_{\mathcal{H}}$ is such that $([\![\mathrm{s}]\!](F) \cap \mathbb{F}) \in [\![\mathrm{s}]\!]_{\mathcal{H}}(\{F\})$
2. $[\![.]\!]_{\mathcal{H}}$ is **adapted for abstract interpretation**, *i.e.*, can be over-approximated in an inductive manner

Hypercollecting semantics

Hypercollecting semantics for expressions:

$$\begin{array}{rccl} [\![\mathrm{e}]\!]_{\mathcal{H}}: & \mathcal{P}(\mathcal{P}(\mathbb{F})) & \longrightarrow & \mathcal{P}(\mathcal{P}(\mathbb{V})) \\ & \mathcal{E} & \longmapsto & \{\{[\![\mathrm{e}]\!](m_1) \mid (m_0, m_1) \in F\} \mid F \in \mathcal{E}\} \end{array}$$

Hypercollecting semantics of tests:

$$\begin{array}{rccl} [\![\mathrm{e}]\!]_{\mathcal{H},\mathrm{test}}: & \mathcal{P}(\mathcal{P}(\mathbb{F})) & \longrightarrow & \mathcal{P}(\mathcal{P}(\mathbb{F})) \\ & \mathcal{E} & \longmapsto & \{\{(m_0, m_1) \in F \mid [\![\mathrm{e}]\!](m_1) = \mathsf{true}\} \mid F \in \mathcal{E}\} \end{array}$$

Hypercollecting semantics of commands:

$$\begin{array}{rcl} [\![\mathrm{e}]\!]_{\mathcal{H}}: \quad \mathcal{P}(\mathcal{P}(\mathbb{F})) & \longrightarrow & \mathcal{P}(\mathcal{P}(\mathbb{F})) \\ [\![\mathrm{x} := \mathrm{e}]\!]_{\mathcal{H}}(\mathcal{E}) & = & \{\{(m_0, m_1[\mathrm{x} \mapsto \mathrm{v}]) \mid (m_0, m_1) \in F \\ & & \quad \wedge [\![\mathrm{e}]\!](m_1) = \mathrm{v}\} \mid F \in \mathcal{E}\} \\ [\![\mathbf{skip}]\!]_{\mathcal{H}}(\mathcal{E}) & = & \mathcal{E} \\ [\![\mathrm{s}_0; \mathrm{s}_1]\!]_{\mathcal{H}}(\mathcal{E}) & = & [\![\mathrm{s}_1]\!] \circ [\![\mathrm{s}_0]\!](\mathcal{E}) \\ [\![\mathbf{if}(\mathrm{e}_0)\ \mathrm{s}_1\ \mathbf{else}\ \mathrm{s}_2]\!]_{\mathcal{H}}(\mathcal{E}) & = & \{[\![\mathrm{s}_1]\!] \circ [\![\mathrm{e}_0]\!]_{\mathcal{H},\mathrm{test}}(F) \\ & & \quad \cup [\![\mathrm{s}_2]\!] \circ [\![\neg \mathrm{e}_0]\!]_{\mathcal{H},\mathrm{test}}(F) \mid F \in \mathcal{E}\} \\ [\![\mathbf{while}(\mathrm{e}_0)\ \mathrm{s}_1]\!]_{\mathcal{H}}(\mathcal{E}) & = & [\![\mathrm{e}]\!]_{\mathcal{H},\mathrm{test}}(\mathbf{lfp}_F\ G_{\mathcal{H}}) \\ & & \text{where } G_{\mathcal{H}} = [\![\mathbf{if}(\mathrm{e}_0)\ \mathrm{s}_1\ \mathbf{else\ skip}]\!]_{\mathcal{H}} \end{array}$$

base semantics

Hypercollecting semantics

Instantiation:

- starting from $\Delta_{\mathbb{M}} = \{\delta_M \mid M \in \mathcal{P}(\mathbb{M})\} = \{\{(m, m) \mid m \in M\} \mid M \in \mathcal{P}(\mathbb{M})\}$
- then, $[\![\mathrm{s}]\!]_{\mathcal{H}}(\Delta_{\mathbb{M}}) \in \mathcal{P}(\mathcal{P}(\mathbb{F}))$ collects the set of **all sets of runs of** s, described by a pair made of an input memory and an output memory
- each of the hypercollecting semantics inputs such a set of sets of pairs

Induction:

- $[\![\mathrm{s}]\!]_{\mathcal{H}}$ is defined by case analysis of s **but its definition is not exactly done by induction**
- but we can prove **by induction**
  1. that it is monotone
  2. the inclusion

set　　　　　　singleton of 1 set

$$([\![\mathrm{s}]\!](F) \cap \mathbb{F}) \in [\![\mathrm{s}]\!]_{\mathcal{H}}(\{F\})$$

- the combination of these properties opens up inductive approximation

Dependence abstraction

We now set up an abstraction for $[\![\mathrm{s}]\!]_{\mathcal{H}}(\Delta_{\mathbb{M}}) \in \mathcal{P}(\mathcal{P}(\mathbb{F}))$, that describes **dependences** between inputs and outputs.

Agreement relation: if $X \subseteq \mathbb{X}$, the equivalence relation $(\equiv_X) \subseteq \mathbb{M} \times \mathbb{M}$ is defined by

$$m_0 \equiv_X m_1 \stackrel{\text{def}}{\Longleftrightarrow} \forall \mathrm{x} \in X,\ m_0(\mathrm{x}) = m_1(\mathrm{x})$$

Dependence abstraction

We let the **dependence abstract domain** be $\mathbb{D}^\sharp_{\mathrm{dep}} = \mathbb{X} \longrightarrow \mathcal{P}(\mathbb{X})$ with the pointwise inclusion ordering, with the **following concretization function**:

$$\begin{array}{rcll}\gamma_{\mathrm{dep}}: & \mathbb{D}^\sharp_{\mathrm{dep}} & \longmapsto & \mathcal{P}(\mathcal{P}(\mathbb{F})) \\ & d & \longrightarrow & \{R \in \mathcal{P}(\mathbb{F}) \mid \forall (m_0, m_1), (m_0', m_1') \in R, \\ & & & \quad \forall \mathrm{x} \in \mathbb{X},\ m_0 \equiv_{d(\mathrm{x})} m_0' \Longrightarrow m_1 \equiv_{\{\mathrm{x}\}} m_1'\}\end{array}$$

Contraposition: when a pair of executions lead to **distinct outputs**, there must be **disagreement in at least some of the dependence inputs**

Dependence abstraction: example

Back to some examples related to non-interference...

$[\![s == 7]\!]^{\#}_{dep}(d) = d(s)$

Program 1:

```
// ... as before, s stores the secret
if( s = 7 )
  i = 1;
else
  i = -1;
```

$[\![\cdot]\!]^{\#}_{d}(\ ): i \mapsto \emptyset$

$[\![\cdot]\!]^{\#}_{d}$

$d \mapsto \{i\}$

$i \mapsto d\{s\}$

non-interference: **violated**

Dependency:

$$i \longmapsto \{s\}$$

Indeed, modifying s may cause distinct i outputs

Program 3:

```
x = 0 * s;
```

non-interference: **satisfied**

Dependency:

$$x \longmapsto \emptyset$$

Indeed, s ends up being 0 regardless...

Non-interference

Non-interference holds if and only if no public variable depends on a secret.

Dependence analysis of expressions

Principle of the dependency analysis of expressions:

- to be used for the analysis of commands
  e.g., assignment command $\mathrm{x} = \mathsf{e}$
  new dependency of x: whatever may change the result of e
- compute an over-approximation of the set of the variables that may make the evaluation result change

Definition of $[\![\mathsf{e}]\!]^\sharp_{\mathrm{dep}} \in \mathbb{D}^\sharp_{\mathrm{dep}} \longrightarrow \mathcal{P}(\mathbb{X})$:

$$
\begin{array}{rcl}
[\![\mathrm{v}]\!]^\sharp_{\mathrm{dep}}(d) & = & \emptyset \\
[\![\mathrm{x}]\!]^\sharp_{\mathrm{dep}}(d) & = & d(\mathrm{x}) \\
[\![\mathsf{e}_0 \oplus \mathsf{e}_1]\!]^\sharp_{\mathrm{dep}}(d) & = & [\![\mathsf{e}_0]\!]^\sharp_{\mathrm{dep}}(d) \dot{\cup} [\![\mathsf{e}_1]\!]^\sharp_{\mathrm{dep}}(d)
\end{array}
$$

$[\![e * 0]\!](d) = \emptyset$

It is **approximate**:

- expression $\mathrm{x} * 0$ does not depend on x in the concrete
- but $[\![\mathrm{x} * 0]\!]^\sharp_{\mathrm{dep}} = \{\mathrm{x}\}$

Soundness of the analysis of expressions

The analysis of expressions is sound in the following sense:

Soundness of the analysis of expressions

Given an expression e and an element $d \in \mathbb{D}^{\sharp}_{\mathrm{dep}}$, then:

$$\forall R \in \gamma_{\mathrm{dep}}(d),\ \forall (m_0, m_1), (m_0', m_1') \in R,$$
$$m_0 \equiv_{[\![\mathrm{e}]\!]^{\sharp}_{\mathrm{dep}}(d)} m_0' \Longrightarrow [\![\mathrm{e}]\!](m_1) = [\![\mathrm{e}]\!](m_1')$$

- the proof proceeds by **induction** over the syntax of expressions
- **example 1**:
  let us assume that e is $\mathtt{x} + \mathtt{y}$
  and that d is $\mathtt{x} \mapsto \{\mathtt{x}\}, \mathtt{y} \mapsto \{\mathtt{x}, \mathtt{t}\}, \mathtt{t} \mapsto \{\mathtt{z}\}$:
  then, $[\![\mathrm{e}]\!]^{\sharp}_{\mathrm{dep}}(d) = \{\mathtt{x}, \mathtt{t}\}$ (this result is **precise**)
- **example 2**:
  let us assume that e is $0 * \mathtt{x}$
  and that d is $\mathtt{x} \mapsto \{\mathtt{x}\}, \ldots$:
  then, $[\![\mathrm{e}]\!]^{\sharp}_{\mathrm{dep}}(d) = \{\mathtt{x}\}$ (this result is **imprecise**)

$\{x\} \cup \{x, t\}$
$\downarrow$
$\{x, t\}$

Dependence analysis of commands

Principle:

- define a function $[\![\mathrm{s}]\!]^\sharp_{\mathrm{dep}} : \mathbb{D}^\sharp_{\mathrm{dep}} \longrightarrow \mathbb{D}^\sharp_{\mathrm{dep}}$ by induction over the syntax of statements
- ensure **soundness condition**

$$[\![\mathrm{s}]\!]_{\mathcal{H}} \circ \gamma_{\mathrm{dep}} \subseteq \gamma_{\mathrm{dep}} \circ [\![\mathrm{s}]\!]^\sharp_{\mathrm{dep}}$$

$$\begin{tikzcd} d \arrow[r, "{[\![s]\!]^\sharp_{dep}}"] \arrow[d, "\gamma_{dep}"'] & \cdot \arrow[d, "\gamma_{dep}"] \\ \cdot \arrow[r, "{[\![s]\!]_{\mathcal{H}}}"'] \arrow[ur, phantom, "\subseteq"] & \cdot \end{tikzcd}$$

- apply $[\![\mathrm{s}]\!]^\sharp_{\mathrm{dep}}$ to $d_{\mathrm{id}} = \lambda \mathrm{x} \in \mathbb{X} \cdot \{\mathrm{x}\}$ (note that $\Delta_{\mathbb{M}} \subseteq \gamma_{\mathrm{dep}}(d_{\mathrm{id}})$)

Analysis of skip commands: $[\![\mathbf{skip}]\!]^\sharp_{\mathrm{dep}}(d) = d$

- since the concrete semantics is also the identity function

Analysis of assignment commands, based on the previously defined $[\![\mathrm{e}]\!]^\sharp_{\mathrm{dep}}$:

$$[\![\mathrm{x} = \mathrm{e}]\!]^\sharp_{\mathrm{dep}}(d) = \left\{ \begin{array}{rcl} \mathrm{x} & \longmapsto & [\![\mathrm{e}]\!]^\sharp_{\mathrm{dep}}(d) \\ \mathrm{y} \neq \mathrm{x} & \longmapsto & d(\mathrm{y}) \end{array} \right.$$

Analysis of sequences: $[\![\mathrm{s}_0; \mathrm{s}_1]\!]^\sharp_{\mathrm{dep}}(d) = [\![\mathrm{s}_1]\!]^\sharp_{\mathrm{dep}} \circ [\![\mathrm{s}_0]\!]^\sharp_{\mathrm{dep}}(d_0)$

- since the concrete semantics is also a composition

Dependence analysis of condition commands

Dependences induced by **condition command if**(e_0) s_1 **else** s_2:

1. dependences in assignments in $\mathrm{s}_1, \mathrm{s}_2$ as before
2. any variable modified in either s_1 or s_2 **also depends on the condition** e_0

Modified variables $\mathcal{M}(\mathrm{s}) \in \mathcal{P}(\mathbb{X})$:

$$\begin{array}{rcl} \mathcal{M}(\mathrm{x} := \mathrm{e}) & = & \{\mathrm{x}\} \\ \mathcal{M}(\mathbf{skip}) & = & \emptyset \\ \mathcal{M}(\mathrm{s}_0; \mathrm{s}_1) & = & \mathcal{M}(\mathrm{s}_0) \cup \mathcal{M}(\mathrm{s}_1) \end{array} \qquad \begin{array}{rcl} \mathcal{M}(\mathbf{if}(\mathrm{e}_0)\ \mathrm{s}_1\ \mathbf{else}\ \mathrm{s}_2) & = & \mathcal{M}(\mathrm{s}_0) \cup \mathcal{M}(\mathrm{s}_1) \\ \mathcal{M}(\mathbf{while}(\mathrm{e}_0)\ \mathrm{s}_1) & = & \mathcal{M}(\mathrm{s}_1) \end{array}$$

Dependency analysis of condition statement $\mathrm{s} ::= \mathbf{if}(\mathrm{e}_0)\ \mathrm{s}_1\ \mathbf{else}\ \mathrm{s}_2$:

- we let $d' = \llbracket \mathrm{s}_1 \rrbracket^\sharp_{\mathrm{dep}} \dot{\cup} \llbracket \mathrm{s}_2 \rrbracket^\sharp_{\mathrm{dep}}$ (pointwise union)
  $\hookrightarrow (d)$ $\hookrightarrow (d)$
- analysis function:

$$\llbracket \mathrm{s} \rrbracket^\sharp_{\mathrm{dep}}(d) = \lambda(\mathrm{x} \in \mathbb{X}) \cdot \begin{cases} d'(\mathrm{x}) \cup \llbracket \mathrm{e}_0 \rrbracket^\sharp_{\mathrm{dep}}(d) & \text{if } \mathrm{x} \in \mathcal{M}(\mathrm{s}_1) \cup \mathcal{M}(\mathrm{s}_2) \\ d'(\mathrm{x}) & \text{otherwise} \end{cases}$$

Case of loops: apply **standard fixpoint techniques**, left as an exercise

Soundness of the analysis of commands

Analysis soundness

For all statement, we have:

1. **soundness of the abstract semantics**:
$$[\![\mathbf{s}]\!]_{\mathcal{H}} \circ \gamma_{\mathrm{dep}} \subseteq \gamma_{\mathrm{dep}} \circ [\![\mathbf{s}]\!]^{\sharp}_{\mathrm{dep}}$$
2. **soundness of the analysis**:
$$[\![\mathbf{s}]\!](\delta_{\mathbb{M}}) \in \gamma_{\mathrm{dep}} \circ [\![\mathbf{s}]\!]^{\sharp}_{\mathrm{dep}}(d_{\mathrm{id}})$$

1. proof by induction over the syntax
2. composing inclusions:

$\{\delta_{\mathbb{M}}\} \subseteq \Delta_{\mathbb{M}}$

$$\begin{array}{rcl} [\![\mathbf{s}]\!](\delta_{\mathbb{M}}) & \in & [\![\mathbf{s}]\!]_{\mathcal{H}}(\{\delta_{\mathbb{M}}\}) \\ & \subseteq & [\![\mathbf{s}]\!]_{\mathcal{H}}(\Delta_{\mathbb{M}}) \\ & \subseteq & [\![\mathbf{s}]\!]_{\mathcal{H}} \circ \gamma_{\mathrm{dep}}(d_{\mathrm{id}}) \\ & \subseteq & \gamma_{\mathrm{dep}} \circ [\![\mathbf{s}]\!]^{\sharp}_{\mathrm{dep}}(d_{\mathrm{id}}) \end{array}$$

Dependence analysis: example implicit flows

did

$$\{x \mapsto \{x\}, y \mapsto \{y\}, s \mapsto \{s\}, z \mapsto \{z\}\}$$

$z = y - 1 + x;$

$$\{x \mapsto \{x\}, y \mapsto \{y\}, s \mapsto \{s\}, z \mapsto \{x, y\}\}$$

$x = s * s + 8;$

$$\{x \mapsto \{s\}, y \mapsto \{y\}, s \mapsto \{s\}, z \mapsto \{x, y\}\}$$

$y = x + 1;$

$$\{\underbrace{x \mapsto \{s\}, y \mapsto \{s\}}_{\text{flow}}, s \mapsto \{s\}, \underbrace{z \mapsto \{x, y\}}_{\text{no-flow}}\}$$

Information flows

- There are **information flows from** s **to** x **and to** y.
- There is **no information flow from** s **to** z.

Dependence analysis: example implicit flows

$$\{x \mapsto \{x\}, y \mapsto \{y\}, s \mapsto \{s\}\}$$

$x = s * s + 8;$

$$\{x \mapsto \{s\}, y \mapsto \{y\}, s \mapsto \{s\}\}$$

$\textbf{if}(x > 0)\,\{$

$$\{x \mapsto \{s\}, y \mapsto \{y\}, s \mapsto \{s\}\}$$

$\quad y = y + 1;$

$$\{x \mapsto \{s\}, y \mapsto \{y\}, s \mapsto \{s\}\}$$

$\}\,\textbf{else}\,\{$

$$\{x \mapsto \{s\}, y \mapsto \{y\}, s \mapsto \{s\}\}$$

$\quad y = y - 1;$

$$\{x \mapsto \{s\}, y \mapsto \{y\}, s \mapsto \{s\}\}$$

$\}$

$$\{x \mapsto \{s\}, y \mapsto \{s, y\}, s \mapsto \{s\}\}$$

Information flows

There are **information flows from** s **to** x **(explicit) and to** y **(implicit)**.

Outline

Another informal proof principle

We look again at the definition of non-interference:

Non-interference

Program P satisfies the **non-interference** property defined by $\mathbb{X}_{\mathrm{pub}}/\ell_{\dashv}, \mathbb{X}_{\mathrm{sec}}/\ell_{\vdash}$ if and only if for all memory states $m_0, m_1 \in \mathbb{M}$,

$$(\forall \mathrm{x} \in \mathbb{X}_{\mathrm{pub}},\ m_0(\mathrm{x}) = m_1(\mathrm{x})) \Longrightarrow (\forall \mathrm{x} \in \mathbb{X}_{\mathrm{pub}},\ [\![P]\!]_{\mathcal{F}[\ell_{\vdash},\ell_{\dashv}]}(m_0)(\mathrm{x}) = [\![P]\!]_{\mathcal{F}[\ell_{\vdash},\ell_{\dashv}]}(m_1)(\mathrm{x}))$$

assertion

Intuition:

- we **run the program twice**, with two states that differ only in the value of one secrete variable
- **if the outputs agree** for all such pairs of runs, then **non-interference** is satisfied

We can turn this **into a symbolic composition**, to allow for the non-interference to be verified.

Proof by self-composition

Notation: to build self-composition, we need to make variables explicit

- we write $P[\mathrm{x}, \mathrm{y}]$ for a program that is defined over variables x, y, even though it may use only some of these;
- for example, we may let $P[\mathrm{x}, \mathrm{y}, \mathrm{z}]$ stand for program **while**$(\mathrm{x} \leq \mathrm{y})\{\mathrm{x} = \mathrm{x} + 1\}$ (z is included even though it is not used in the program)

Definition: proof by self-composition

Let $P[\mathrm{s}_0, \ldots, \mathrm{s}_k, \mathrm{x}_0, \ldots, \mathrm{x}_l]$ be a deterministic program, where $\mathbb{X}_{\mathrm{sec}} = \{\mathrm{s}_0, \ldots, \mathrm{s}_k\}$ and $\mathbb{X}_{\mathrm{pub}} = \{\mathrm{x}_0, \ldots, \mathrm{x}_l\}$. We let $\mathrm{s}'_0, \ldots, \mathrm{s}'_k, \mathrm{x}'_0, \ldots, \mathrm{x}'_l$ be **fresh** variables. We let $Q[\mathrm{s}_0, \ldots, \mathrm{s}_k, \mathrm{x}_0, \ldots, \mathrm{x}_l, \mathrm{s}'_0, \ldots, \mathrm{s}'_k, \mathrm{x}'_0, \ldots, \mathrm{x}'_l]$ be:

$$\begin{array}{l}
\textbf{assume}(\mathrm{x}_0 == \mathrm{x}'_0); \ldots; \textbf{assume}(\mathrm{x}_l == \mathrm{x}'_l); \\
P[\mathrm{s}_0, \ldots, \mathrm{s}_k, \mathrm{x}_0, \ldots, \mathrm{x}_l]; \\
P'[\mathrm{s}'_0, \ldots, \mathrm{s}'_k, \mathrm{x}'_0, \ldots, \mathrm{x}'_l]; \\
\textbf{assert}(\mathrm{x}_0 == \mathrm{x}'_0); \ldots; \textbf{assert}(\mathrm{x}_l == \mathrm{x}'_l);
\end{array}$$

Then, $P[\ldots]$ **satisfies non-interference if and only if** $Q[\ldots]$ **satisfies the final assertion**.

↳ safety

Proof by self-composition

Principle: **reduce a security question to a safety question but for a different program**

- initial question: **is P secure ?**
- reduced question: **is Q safe ?** (where Q is defined from P)
- then, classical analysis techniques for safety apply

Specific issues:

- **termination**:
  if P may not terminate, the observation of termination or non-termination may reveal information on the secret
- **non-determinism**:
  if P may contain some non-determinism, the final assertion of Q may fail even when the non-interference is satisfied

Taking these into account **will require more care**.

Examples

A simple case:

Initial program:

```
x = 8 * y + 2;
s = x + s;
```

Verification of the assertion by static analysis:
exercise: which abstract domain ?

Transformed program:

```
assume( x0 == x1 );
x0 = 8 * y0 + 2;
s0 = x0 + s0;
x1 = 8 * y1 + 2;
s1 = x1 + s1;
assert( x0 == x1 );
```

assume(y0==y1

A subtle case to rule out **deceptive implicit flows**:

Initial program:

```
if( s == 1 ) x = s;
else x = 1;
```

$\equiv x = 1$

$x \mapsto \{s\}$

```
assume( x0 == x1 );
if( s0 == 1 ) x0 = s0;
else x0 = 1;
if( s1 == 1 ) x1 = s1;
else x1 = 1;
assert( x0 == x1 );
```

Outline

Main points to remember

Security properties **are a separate class of properties**:

- expressing the property requires **quantifying over pairs of executions**
- **hyperproperties** ⊃ hypersafety ⊃ **2-safety**
  many important security properties are 2-safety...

Static analysis with respect to hyperproperties:

- **dependence analysis** has to be proved with respect to a **specific semantics**, which can talk about pairs of executions
- **deceptive implicit flows**: conditions

Self-composition:
technique based on the **reduction** to another property

K-safety

Assignment: proofs and paper reading

Recognizing Safety and Liveness.
Bowen Alpern and Fred B. Schneider.
In Distributed Computing, Springer, 1987.

Hyperproperties.
Michael Clarkson and Fred B. Schneider.
In CSF 2008, IEEE, 2008.

Hypercollecting semantics and its application to static analysis of information flow.
Mounir Assaf, David A. Naumann, Julien Signoles, Eric Totel, Frédéric Tronel.
In POPL'17, pages 874–887, 2017.

Secure Information Flow by Self-Composition.
Gilles Barthe, Pedro R. D'Argenio, Tamara Rezk.
In CSFW 2004: 100-114