Formal Methods for Security and Privacy (SS 2022) Univ-Prof. Dr. Matteo Maffei

Final Exam

Date: 29/06/2022 TU Wien

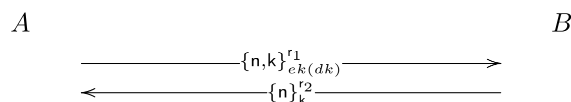
- The exam takes 180 minutes. You can get at most 100 points.
- Your solutions can be handwritten and then scanned or photographed, or directly typed up.
  - Upload a single file (either a single pdf, or a zip archive containing multiple images (jpeg, png)) to the TUWEL assignment.
  - Make sure that the result is legible.
  - Your file may not be bigger than 256MB.
  - Your solution must be handed in **before 13:00 on the day of the exam**.
- **You must work on the exam alone.** You may use available resources (for example lecture slides), but **you must not communicate with anyone** except the lecturers. Everything you hand in must be written and created by you and you must be able to explain your solution. You will be required to confirm this when you upload your file. If plagiarism is detected, all of the parties involved (both committing and aiding) will be held accountable.
- During the exam you are required to be connected to the following Zoom call with your camera and microphone on: `https://tuwien.zoom.us/j/91250537864`
- **Write legibly and be concise**. It is in your best interest that we understand your answers. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it.
- Please make sure your name and matriculation number are written on the document you hand in.
- Good luck!

Problem	1	2	3	**Total**
Points	30	40	30	100

Problem 1: Computational model & Bana-Comon Logic (30 points)

NB: You will find the definition of relevant security games and BC-logic axioms in figures 1, 2 and 3. In the following we assume keys and nonces have length η.

Alice and Bob decide to use the following protocol P:



That is, Alice sends a fresh nonce n and a key k to Bob encrypted with his public key. Then Bob answers by encrypting Alice's n that he just received with a symmetric encryption scheme using k. Both encryption schemes are properly randomized with fresh nonces and are IND-CCA2 and IND-CPA secure. For simplicity, we will assume there is only one session of this protocol and that Alice, Bob and Eve are the only ones participating.

The attacker wants to know n.

To prove the secrecy of n, we will do a game-based proof. This means proving that we can move from the initial situation to a trivially secure one in an indistinguishable way.

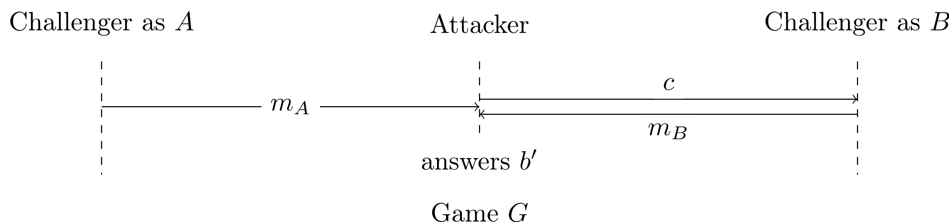
From now on we assume the attacker plays against a simulator who, given an instance of the protocol (that is a $(\mathsf{n}, \mathsf{k}, \mathsf{r}_1, \mathsf{r}_2, dk)$), follows perfectly the protocol P. We call this game S_0.

a) **(9 points)**

Our first step is to blur Alice message. To this end, we build a new similator that only sends $m_A = \{0^{2\eta}\}^{\mathsf{r}_1}_{ek(dk)}$ as Alice and answers $\{\mathsf{n}\}^{\mathsf{r}_2}_{\mathsf{k}}$ as Bob's when the input c is m_A, otherwise it behaves as before. That is, it answers $\{\pi_1(\mathsf{dec}(c, dk))\}^{\mathsf{r}_2}_{\pi_2(\mathsf{dec}(c, dk_B))}$ when the decription and projections succeed, or it errors out as Bob would have otherwise.

We call this new game S_1. We want to show that the attacker cannot distinguish between S_0 and S_1.

To clarify the proof, let us consider yet another cryptographic game:



Game G

where

$$m_A = \begin{cases} \{\mathsf{n}, \mathsf{k}\}^{\mathsf{r}_1}_{ek(dk)} & \text{if } b = 0 \\ \{0^{2\eta}\}^{\mathsf{r}_1}_{ek(dk)} & \text{else} \end{cases}$$

$$m_B = \begin{cases} \{\mathsf{n}\}^{\mathsf{r}_2}_{\mathsf{k}} & \text{if } c = m_A \\ \{\pi_1(\mathsf{dec}(c, dk))\}^{\mathsf{r}_2}_{\pi_2(\mathsf{dec}(c, dk_B))} & \text{else if decription and projection succeed} \\ \mathtt{err} & \text{else} \end{cases}$$

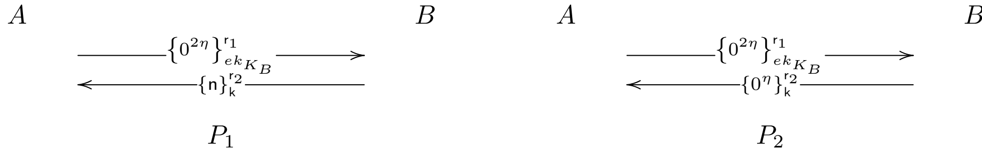
with n, k, r_1, r_2, dk are picked at random. The attacker wins the game if $b' = b$.

Show that if there is an attacker that breaks game G, then we can build an attacker that breaks IND-CCA2.

We conclude that $\mathbf{Adv}(t; S_0, S_1)$ is negligible.

b) **(5 + 14 points)** The next step is to blur Bob's message. In a similar fashion, we build a new simulator that behaves like in S_1 except it answers with $\{0^{\eta}\}^{\mathsf{r}_2}_{\mathsf{k}}$ when $c = m_A$. We call this new game S_2.

This time, we decide to use BC-logic. First, let's show that the two following protocols are observationally equivalent:



(i) Model P_1 and P_2 as partially ordered sets of steps.

(ii) Prove that for all timepoints τ, $\phi_{P_1}(\tau) \sim \phi_{P_2}(\tau)$ using the BC-logic

NB: To keep the proof tree's size manageable, feel free to apply multiple times the same rule and write it only once. Similarily, you can omit the uses of the DUP rule. Simple margin notes are also enough to justify the subterm queries.

c) **(2 point)**

Furthermore, we can prove that

$$\mathbf{Adv}(t; S_1, S_2) \leq \mathbf{Adv}(t; P_1, P_2) \qquad (*)$$

.

Conclude then that the protocol is secure.

Base indistinguishability rules

$$\text{REFL}\ \frac{}{\Gamma \vdash \vec{u} \sim \vec{u}} \qquad \text{AX}\ \frac{}{\Gamma, \vec{u} \sim \vec{v} \vdash \vec{u} \sim \vec{v}} \qquad \text{DUP}\ \frac{\Gamma \vdash \xi, \vec{u} \sim \xi', \vec{v}}{\Gamma \vdash \xi, \xi, \vec{u} \sim \xi', \xi', \vec{v}} \qquad \text{ENRICH}\ \frac{\Gamma \vdash \xi, \vec{u} \sim \xi', \vec{v}}{\Gamma \vdash \vec{u} \sim \vec{v}}$$

$$\text{FA}\ \frac{\Gamma \vdash t_1, \ldots, t_n, \vec{u} \sim t'_1, \ldots, t'_n, \vec{v}}{\Gamma \vdash f(t_1, \ldots, t_n), \vec{u} \sim f(t'_1, \ldots, t'_n), \vec{v}} \qquad \text{SUP}\ \frac{\Gamma \vdash t \doteq t' \qquad \Gamma \vdash \vec{u}\{t \to t'\} \sim \vec{v}\{t \to t'\}}{\Gamma \vdash \vec{u} \sim \vec{v}}$$

Let S be the set of steps, induction is defined as:

$$\text{INDUCTION}\ \frac{\begin{array}{c}\Gamma \vdash \phi_{\mathcal{P}_1}(\text{init}), \vec{u} \sim \phi_{\mathcal{P}_2}(\text{init}), \vec{v} \\ \Gamma, \phi_{\mathcal{P}_1}(\text{pred}(\tau)), \vec{u} \sim \phi_{\mathcal{P}_2}(\text{pred}(\tau)), \vec{v} \vdash \phi_{\mathcal{P}_1}(\tau), \vec{u} \sim \phi_{\mathcal{P}_2}(\tau), \vec{v}, \text{ for all } \tau \in S\end{array}}{\Gamma \vdash \forall \tau, \phi_{\mathcal{P}_1}(\tau), \vec{u} \sim \phi_{\mathcal{P}_2}(\tau), \vec{v}}$$

Cryptographic rules

$$\text{IND-CPA}\ \frac{\Gamma \vdash \vec{v} \sim \vec{u}, C\left[\left\{0^{|m|}\right\}^{\mathsf{r}}_{\mathsf{k}}\right] \qquad \mathsf{r}, \mathsf{k} \not\sqsubseteq m, \vec{u}}{\Gamma \vdash \vec{v} \sim \vec{u}, C\left[\{m\}^{\mathsf{r}}_{\mathsf{k}}\right]} \qquad \text{FRESH}\ \frac{\Gamma \vdash \vec{u} \sim \vec{v} \qquad \mathsf{n}, \mathsf{m} \not\sqsubseteq \vec{u}, \vec{v}}{\Gamma \vdash \vec{u}, \mathsf{n} \sim \vec{v}, \mathsf{m}}$$

Frame

$$\phi_{\mathcal{P}}(\tau) \doteq (\text{condition}(\tau), \text{message}(\tau), \phi_{\mathcal{P}}(\text{pred}(\tau)))$$

Figure 1: Relevant subset of BC-Logic rules

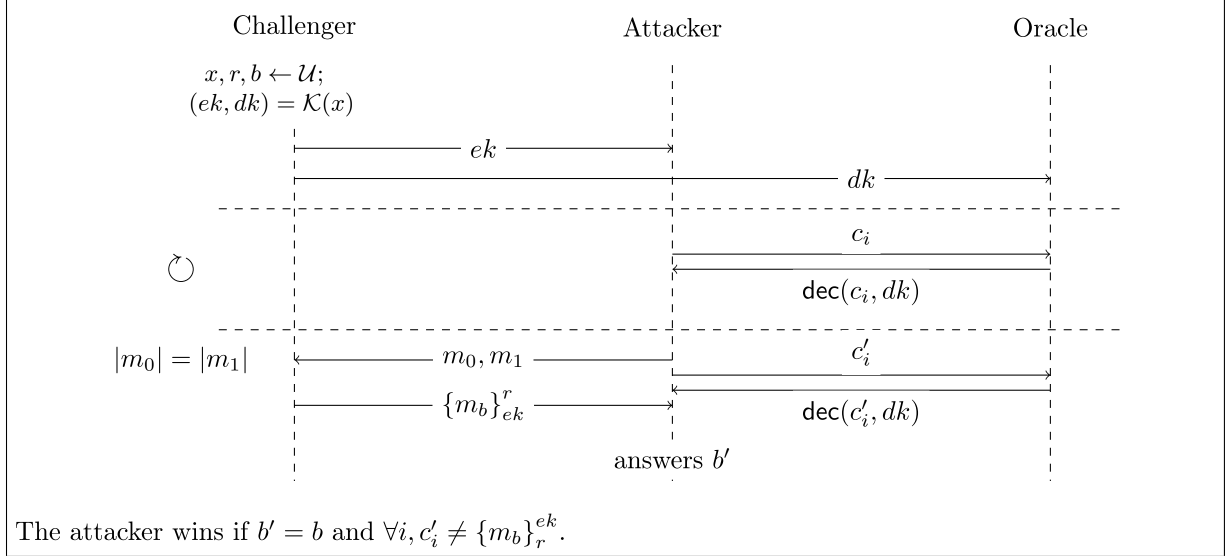


Figure 2: IND-CCA2 cryptographic game

We define the **advantage** as

$$\mathbf{Adv}(t; S'_0, S'_1) = \max_{\mathcal{A}} |\mathbf{P}(\mathcal{A}(t) = 1 | S'_0) - \mathbf{P}(\mathcal{A}(t) = 1 | S'_1)| \tag{**}$$

Note that

$$\mathbf{Adv}(t; A, C) \leq \mathbf{Adv}(t; A, B) + \mathbf{Adv}(t; B, C) \tag{***}$$

Figure 3: Reminder

Problem 2: Information Flow (40 points)

Let $\mathbb{L} = \{\{L, H\}, \sqsubseteq, \sqcup\}$ be a security lattice, where $\sqsubseteq$ denotes the partial order and where $\sqcup$ denotes the least upper bound (remember that e.g. $L \sqsubseteq H$ or $L \sqcup H = H$). We then define a simple WHILE language where the set of variables $\mathbb{V}$ is partitioned into two disjoint sets, $\mathbb{V}_H$, the set of high (or secret) variables, and $\mathbb{V}_L$, the set of low (or public) variables.

$$\begin{array}{llll} \textbf{Expr} & ::= & n & n \in \mathbb{Z} \\ & \mid & x & x \in \mathbb{V}_H \uplus \mathbb{V}_L \\ & \mid & \textbf{Expr} \text{ op } \textbf{Expr} & \text{op} \in \{+, -, =, <, \dots\} \end{array} \qquad \begin{array}{lll} \textbf{Cmd} & ::= & x := \textbf{Expr} \\ & \mid & \texttt{if } \textbf{Expr} \texttt{ then } \textbf{Cmd} \texttt{ else } \textbf{Cmd} \\ & \mid & \texttt{while } \textbf{Expr} \texttt{ do } \textbf{Cmd} \\ & \mid & \textbf{Cmd} \,;\, \textbf{Cmd} \end{array}$$

In this exercise, we present a simple information flow type system and prove it enforces a semantic non-interference property on well-typed WHILE programs.

Typing judgment for expressions: for $e \in \textbf{Expr}$ and $\tau \in \{L, H\}$, $\vdash e : \tau$ means that the expression e depends only on variables of level τ or **lower**.

$$\text{CONST}\ \frac{}{\vdash n : L} \qquad \text{VAR}\ \frac{\tau_x = \ell \text{ such that } x \in \mathbb{V}_\ell}{\vdash x : \tau_x}$$

$$\text{OP}\ \frac{\vdash e_1 : \tau \qquad \vdash e_2 : \tau}{\vdash e_1 \text{ op } e_2 : \tau} \qquad \text{EXPR-SUBTYPE}\ \frac{\vdash e : \tau_1 \qquad \tau_1 \sqsubseteq \tau_2}{\vdash e : \tau_2}$$

Typing judgment for commands: for $c \in \textbf{Cmd}$ and $\tau \in \{L, H\}$, $\tau \models c$ means that the variables modified by command c are of level τ or **higher**.

$$\text{ASSIGN}\ \frac{\vdash e : \tau_e \qquad \tau_e \sqcup \tau \sqsubseteq \tau_x \qquad \tau_x = \ell \text{ such that } x \in \mathbb{V}_\ell}{\tau \models x := e} \qquad \text{SEQ}\ \frac{\tau \models c_1 \qquad \tau \models c_2}{\tau \models c_1 \,;\, c_2}$$

$$\text{IF}\ \frac{\vdash e : \tau_e \qquad \tau_e \sqcup \tau \models c_1 \qquad \tau_e \sqcup \tau \models c_2}{\tau \models \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2} \qquad \text{WHILE}\ \frac{\vdash e : \tau_e \qquad \tau_e \sqcup \tau \models c}{\tau \models \texttt{while } e \texttt{ do } c}$$

$$\text{CMD-SUBTYPE}\ \frac{\tau_1 \models c \qquad \tau_2 \sqsubseteq \tau_1}{\tau_2 \models c}$$

a) **(8 Points)** Find two examples, one using the $\texttt{if} \cdot \texttt{then} \cdot \texttt{else} \cdot$ construction, one using the $\texttt{while} \cdot \texttt{do} \cdot$ construction, such that:

   i) one is well-typed according to the above type system (show a type derivation);

   ii) one is non-interferent but not well-typed (explain why your example is non-interferent and point out where type-checking necessarily fails).

We want to prove that the type system is indeed ensuring non-interference. To do so, we first need to define the semantics of our language. Let $\sigma \in \textbf{State} = \mathbb{V} \to \mathbb{Z}$ be a mapping from variables to values.

Semantics of expressions: for $e \in \textbf{Expr}$ and $\sigma \in \textbf{State}$, $[\![e]\!]_\sigma = v$ means that evaluating expression e in state σ returns v.

$$[\![n]\!]_\sigma = n \qquad [\![x]\!]_\sigma = \sigma(x) \qquad [\![e_1 \text{ op } e_2]\!]_\sigma = [\![e_1]\!]_\sigma \text{ op } [\![e_2]\!]_\sigma$$

Semantics of commands: for $c \in \textbf{Cmd}$ and $\sigma, \sigma' \in \textbf{Cmd}$, $(c, \sigma) \Downarrow \sigma'$ means that executing c on state σ returns σ'.

$$\frac{}{(x := e, \sigma) \Downarrow \sigma[x \mapsto [\![e]\!]_\sigma]} \qquad \frac{(c_1, \sigma) \Downarrow \sigma' \qquad (c_2, \sigma') \Downarrow \sigma''}{(c_1 \,;\, c_2, \sigma) \Downarrow \sigma''}$$

$$\frac{(c_1, \sigma) \Downarrow \sigma' \qquad [\![e]\!]_\sigma \neq 0}{(\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, \sigma) \Downarrow \sigma'} \qquad \frac{(c_2, \sigma) \Downarrow \sigma' \qquad [\![e]\!]_\sigma = 0}{(\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, \sigma) \Downarrow \sigma'}$$

$$\frac{(c, \sigma) \Downarrow \sigma' \qquad (\texttt{while } e \texttt{ do } c, \sigma') \Downarrow \sigma'' \qquad [\![e]\!]_\sigma \neq 0}{(\texttt{while } e \texttt{ do } c, \sigma) \Downarrow \sigma''} \qquad \frac{[\![e]\!]_\sigma = 0}{(\texttt{while } e \texttt{ do } c, \sigma) \Downarrow \sigma}$$

Definition 1 (Indistinguishability). *For $\sigma_1, \sigma_2 \in$ **State**, $\sigma_1 \sim \sigma_2$ iff $\forall x \in \mathbb{V}_L, \sigma_1(x) = \sigma_2(x)$.*

Definition 2 (Non-interference). *A command $c \in$ **Cmd** is said to be non-interferent iff for all $\sigma_1, \sigma_2 \in$ **State** s.t. $\sigma_1 \sim \sigma_2$, $(c, \sigma_1) \Downarrow \sigma_1'$ and $(c, \sigma_2) \Downarrow \sigma_2'$ implies $\sigma_1' \sim \sigma_2'$.*

Theorem 1 (Type soundness). *Every typeable command (i.e. such that $\exists \tau, \tau \models c$) is non-interferent.*

We want to prove Theorem 1. To this end, we introduce a simpler set of typing rules we prove to be equivalent.

$$\text{CONST}^\star \ \frac{}{\vdash^\star n : \tau} \qquad \text{VAR}^\star \ \frac{\tau_x \sqsubseteq \tau \qquad \tau_x = \ell \text{ such that } x \in \mathbb{V}_\ell}{\vdash^\star x : \tau} \qquad \text{OP}^\star \ \frac{\vdash^\star e_1 : \tau \qquad \vdash^\star e_2 : \tau}{\vdash^\star e_1 \text{ op } e_2 : \tau}$$

$$\text{ASSIGN}^\star \ \frac{\vdash^\star e : \tau_x \qquad \tau \sqsubseteq \tau_x \qquad \tau_x = \ell \text{ such that } x \in \mathbb{V}_\ell}{\tau \models^\star x := e} \qquad \text{SEQ}^\star \ \frac{\tau \models^\star c_1 \qquad \tau \models^\star c_2}{\tau \models^\star c_1 \, ; c_2}$$

$$\text{IF}^\star \ \frac{\vdash^\star e : \tau_e \qquad \tau_e \models^\star c_1 \qquad \tau_e \models^\star c_2 \qquad \tau \sqsubseteq \tau_e}{\tau \models^\star \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2} \qquad \text{WHILE}^\star \ \frac{\vdash^\star e : \tau_e \qquad \tau_e \models^\star c \qquad \tau \sqsubseteq \tau_e}{\tau \models^\star \texttt{while } e \texttt{ do } c}$$

Lemma 1 (Sub-typing property).

- *For all e, τ, τ', $\vdash^\star e : \tau$ and $\tau \sqsubseteq \tau'$ implies $\vdash^\star e : \tau'$.*
- *For all c, τ, τ', $\tau' \models^\star c$ and $\tau \sqsubseteq \tau'$ implies $\tau \models^\star c$.*

b) **(6 Points)** Prove Lemma 1 by induction on the typing judgment. Give a general intuition and elaborate only cases you think are the most important.

Lemma 2 (Type system equivalence).

i)
- *For all e, τ, $\vdash e : \tau$ implies $\vdash^\star e : \tau$.*
- *For all c, τ, $\tau \models c$ implies $\tau \models^\star c$.*

ii)
- *For all e, τ, $\vdash^\star e : \tau$ implies $\vdash e : \tau$.*
- *For all c, τ, $\tau \models^\star c$ implies $\tau \models c$.*

c) **(6 Points)** Using Lemma 1, prove Lemma 2 by induction on the typing judgment. Give a general intuition and elaborate only cases you think are the most important.

Lemma 3 (Low expressions). *For all $e \in$ **Expr**, if $\vdash^\star e : L$, then for all $\sigma_1, \sigma_2 \in$ **State**, $\sigma_1 \sim \sigma_2$ implies $[\![e]\!]_{\sigma_1} = [\![e]\!]_{\sigma_2}$.*

d) **(6 Points)** Prove Lemma 3 by induction on type derivation for e. Give a general intuition and elaborate only cases you think are the most important.

Lemma 4 (Confinement of high commands). *For all $c \in$ **Cmd** and $\sigma, \sigma' \in$ **State**, if $(c, \sigma) \Downarrow \sigma'$ and $H \models^\star c$, then $\sigma \sim \sigma'$.*

e) **(6 Points)** Prove Lemma 4 by induction on the judgement $(c, \sigma) \Downarrow \sigma'$. Give a general intuition and elaborate only cases you think are the most important.

Theorem 2 (Type soundness). *For all $c \in$ **Cmd**, $\tau \in \{L, H\}$ and $\sigma_1, \sigma_2, \sigma_1', \sigma_2' \in$ **State**, if $\sigma_1 \sim \sigma_2$, $(c, \sigma_1) \Downarrow \sigma_1'$, $(c, \sigma_2) \Downarrow \sigma_2'$, and $\tau \models^\star c$, then $\sigma_1' \sim \sigma_2'$.*

f) **(8 Points)** Using Lemma 3 and Lemma 4, prove Theorem 2 by induction on the judgement $(c, \sigma) \Downarrow \sigma'$. Give a general intuition and elaborate only cases you think are the most important. Be careful with the `while` case. Then, conclude about Theorem 1.

Problem 3: Static Analysis (30 points)

Assume the tiny imperative language with the following instructions:

$$\mathsf{PUSH}\ v,\ \mathsf{INPUT}\ k,\ \mathsf{SELECT},\ \mathsf{ADD},\ \mathsf{JUMPif}\ i \qquad\qquad i, v, k \in \mathbb{N}$$

You can think of programs in this language as simple imperative programs that are given a map of inputs, can modify a stack and otherwise can just alter the control flow by conditional jumps. More precisely, the instruction PUSH v pushes the value v to the stack, the instruction INPUT k reads the k-th input value and pushes it to the stack, the instruction SELECT pops three elements from the stack and pushes the second or third, depending on the first, the instruction ADD adds the two top values from the stack, pops them, and writes the result to the stack, and finally JUMPif i reads the top value from the stack, pops it, and jumps to the instruction at program counter i only if the top stack value is not equal to 0.

More formally, we describe the state of a program by a configuration of the form (pc, s, im). Intuitively, such a configuration says that pc is the current program counter, s (a list of natural numbers) is the current stack and im (a function $\mathbb{N} \mapsto \mathbb{N}$) is the input of the program. Note that we write ϵ for the empty stack and $v :: s'$ for a stack with top element v and substack s'.

Next, we can define the semantics of our language in terms of a small-step relation. More precisely, $code \vdash (pc, s, im) \to (pc', s', im')$ means that given a program $code$ (which is just a list of instructions), the configuration changes from (pc, s, im) to (pc', s', im') within processing one instruction. Formally, the small-step relation is defined by the following inference rules:

PUSH
$$\frac{code[pc] = \mathsf{PUSH}\ v}{code \vdash (pc, s, im) \to (pc+1, v :: s, im)}$$

INPUT
$$\frac{code[pc] = \mathsf{INPUT}\ k}{code \vdash (pc, s, im) \to (pc+1, im(k) :: s, im)}$$

SELECT-FIRST
$$\frac{code[pc] = \mathsf{SELECT} \qquad v_1 = 0}{code \vdash (pc, v_1 :: (v_2 :: (v_3 :: s)), im) \to (pc+1, v_2 :: s, im)}$$

SELECT-SECOND
$$\frac{code[pc] = \mathsf{SELECT} \qquad v_1 \neq 0}{code \vdash (pc, v_1 :: (v_2 :: (v_3 :: s)), im) \to (pc+1, v_3 :: s, im)}$$

ADD
$$\frac{code[pc] = \mathsf{ADD}}{code \vdash (pc, v_1 :: (v_2 :: s), im) \to (pc+1, (v_1 + v_2) :: s, im)}$$

JUMPIF-TRUE
$$\frac{code[pc] = \mathsf{JUMPif}\ i \qquad v \neq 0}{code \vdash (pc, v :: s, im) \to (i, s, im)}$$

JUMPIF-FALSE
$$\frac{code[pc] = \mathsf{JUMPif}\ i \qquad v = 0}{code \vdash (pc, v :: s, im) \to (pc+1, s, im)}$$

a) **(6 Points)** Assume the following simple program:

```
 1    INPUT 1
 2    INPUT 4
 3    PUSH 1
 4    INPUT 0
 5    SELECT
 6    JUMPif 11
 7    PUSH 3
 8    ADD
 9    PUSH 1
10    JUMPif 13
11    INPUT 5
12    ADD
```

$$args(x) = \begin{cases} 100 & \text{if } x = 0 \\ 200 & \text{if } x = 1 \\ 0 & \text{otherwise} \end{cases}$$

Starting in the configuration $(1, \epsilon, args)$, which configurations are reachable using the small step semantics? Name the rules (in the right order) that need to be applied to reach the corresponding configurations and give all intermediate configurations.

b) **(6 Points)** An important concept in static analysis is soundness. For a static analysis defined in terms of Horn clause resolution (as we have seen in the lecture) soundness can be defined as follows:

Given an abstraction function α that maps configurations into a state predicate (S) which models the abstract configuration and a function h that defines the Horn clauses describing the abstract semantics, a static analysis is sound if the implication below holds.

$$\mathit{code} \vdash c \to^* c' \implies \forall \Delta \geq \alpha(c).\ \exists \Delta' \geq \alpha(c').\ \Delta, h(\mathit{code}) \vdash \Delta'$$

In prose this states that whenever configuration c executes *code* and reaches configuration c' (within some number of steps) then it also holds that from any abstract configuration Δ that is at least as abstract as $\alpha(c)$ ($\Delta \geq \alpha(c)$) one can logically derive ($\vdash$) an abstract configuration Δ' that is at least as abstract as $\alpha(c')$ ($\Delta' \geq \alpha(c')$) using the Horn clauses $h(\mathit{code})$.

Intuitively, an abstract configuration Δ is at least as abstract as an abstract configuration Δ' if for every fact (predicate application) in Δ' one can find one in Δ that is at least as abstract. Formally $\geq$ on abstract configurations is defined as follows:

$$\Delta \geq \Delta' := \quad \forall f' \in \Delta'.\ \exists f \in \Delta. f \geq_F f'$$

Note that we also need to define what it means for a fact (predicate application) to be as abstract as another ($\geq_F$), since this notion of abstraction is specific to the analysis.

Given these definitions, assume that there are two analyses A and B that check for a security property S. S is defined in terms of certain problematic states (that is: if certain problematic states are not reachable, a program is considered secure). Assume that A is known to be sound, while the soundness of B is unclear. Both analyze a program P. What conclusions can you draw about the soundess of B given the following outcomes:

- A labels P secure, B labels P secure
- A labels P secure, B labels P insecure
- A labels P insecure, B labels P secure
- A labels P insecure, B labels P insecure

Explain your reasoning!

c) **(9 + 9 Points)** In the following, we will present different approaches to defining an abstract analysis for the presented language using the formalism introduced above.

Your task is to state for each of the presented analysis approaches if they satisfy the soundness claim.

- If yes, argue why the soundness claim holds and name (and explain) at least one point where the analysis looses precision. More concretely, give an example program *code* and initial configuration c such that that there is an abstract configuration Δ' that is derivable from $\alpha(c)$, but there is no concrete configuration c' that is reachable from c such that $\alpha(c') = \Delta'$. Provide also all reachable configurations c' explicitly, as well as $h(\mathit{code})$.
- If not, give a counter example to the soundness claim. More precisely give a program *code* and initial configuration c, and an abstract configuration $\Delta \geq \alpha(c)$ such that there is a configuration c' that is reachable from c, but there is no $\Delta' \geq \alpha(c')$ that is derivable from Δ. Provide also c' and $h(\mathit{code})$ explicitly and explain why no such Δ' is derivable.

(i) Predicates:

$$\{\mathsf{S}_i(a, b) \mid i \in \mathbb{N} \wedge a, b \in \mathbb{Z}\}$$

Abstraction relation:

$$\mathsf{S}_i(a_1, b_1) \geq_F \mathsf{S}_i(a_2, b_2) := (a_1 < 0 \vee a_1 = a_2) \wedge (b_1 < 0 \vee b_1 = b_2)$$

Abstraction of configurations:

$$\alpha_1((pc, s, im)) = \begin{cases} \{\mathsf{S}_{pc}(-1, -1)\} & s = \epsilon \\ \{\mathsf{S}_{pc}(v, -1)\} & s = v :: \epsilon \\ \{\mathsf{S}_{pc}(v_1, v_2) & s = v_1 :: (v_2 :: s') \end{cases}$$

Abstract rules:

$$
\begin{aligned}
h_1(code) = &\\
&\{\mathsf{S}_{pc}(a,b) \Rightarrow \mathsf{S}_{pc+1}(v,a) \mid code[pc] = \mathsf{PUSH}\ v\}\\
&\cup \{\mathsf{S}_{pc}(a,b) \Rightarrow \mathsf{S}_{pc+1}(-1,a) \mid code[pc] = \mathsf{INPUT}\ k\}\\
&\cup \{\mathsf{S}_{pc}(a,b), a = 0 \vee a < 0 \Rightarrow \mathsf{S}_{pc+1}(b,-1) \mid code[pc] = \mathsf{SELECT}\}\\
&\cup \{\mathsf{S}_{pc}(a,b), a \neq 0 \Rightarrow \mathsf{S}_{pc+1}(-1,-1) \mid code[pc] = \mathsf{SELECT}\}\\
&\cup \{\mathsf{S}_{pc}(a,b) \wedge a \geq 0 \wedge b \geq 0 \Rightarrow \mathsf{S}_{pc+1}(a+b,-1) \mid code[pc] = \mathsf{ADD}\}\\
&\cup \{\mathsf{S}_{pc}(a,b) \wedge (a < 0 \vee b < 0) \Rightarrow \mathsf{S}_{pc+1}(-1,-1) \mid code[pc] = \mathsf{ADD}\}\\
&\cup \{\mathsf{S}_{pc}(a,b) \wedge a \leq 0 \Rightarrow \mathsf{S}_{pc+1}(b,-1) \mid code[pc] = \mathsf{JUMPif}\ i\}\\
&\cup \{\mathsf{S}_{pc}(a,b) \wedge a \neq 0 \Rightarrow \mathsf{S}_{i}(b,-1) \mid code[pc] = \mathsf{JUMPif}\ i\}
\end{aligned}
$$

(ii) Predicates:

$$\{\mathsf{S}_i(s) \mid i \in \mathbb{N} \wedge s \in \mathcal{L}(\mathbb{N} \cup \{\top\})\}$$

where $\mathcal{L}(\mathbb{N} \cup \{\top\})$ denotes the set of lists over abstract natural numbers (i.e. natural numbers together with a special value $\top$).

Abstraction relation:

$$\mathsf{S}_i(s_1) \geq_F \mathsf{S}_i(s_2) := s_1 \geq_F s_2$$

$$s_1 \geq_F s_2 := \begin{cases} s_1' \geq_F s_2' & \text{if } (s_1 = (v_1 :: s_1')) \wedge (s_2 = (v_2 :: s_2')) \wedge (v_1 = \top \vee v_1 = v_2) \\ \mathit{true} & \text{if } s_1 = \epsilon \wedge s_2 = \epsilon \\ \mathit{false} & \text{otherwise} \end{cases}$$

Abstraction of configurations:

$$\alpha_2((pc, s, im)) = \{\mathsf{S}_{pc}(s)\}$$

Abstract rules:

$$
\begin{aligned}
h_2(code) = &\\
&\{\mathsf{S}_{pc}(s) \Rightarrow \mathsf{S}_{pc+1}(v :: s) \mid code[pc] = \mathsf{PUSH}\ v\}\\
&\cup \{\mathsf{S}_{pc}(s) \Rightarrow \mathsf{S}_{pc+1}(\top :: s) \mid code[pc] = \mathsf{INPUT}\ k\}\\
&\cup \{\mathsf{S}_{pc}(v_1 :: (v_2 :: (v_3 :: s))) \wedge v_1 = 0 \Rightarrow \mathsf{S}_{pc+1}(v_2 :: s) \mid code[pc] = \mathsf{SELECT}\}\\
&\cup \{\mathsf{S}_{pc}(v_1 :: (v_2 :: (v_3 :: s))) \wedge v_1 \neq 0 \Rightarrow \mathsf{S}_{pc+1}(v_3 :: s) \mid code[pc] = \mathsf{SELECT}\}\\
&\cup \{\mathsf{S}_{pc}(v_1 :: (v_2 :: s)) \Rightarrow \mathsf{S}_{pc+1}((v_1 + v_2) :: s) \mid code[pc] = \mathsf{ADD}\}\\
&\cup \{\mathsf{S}_{pc}(v :: s) \wedge (v = 0 \vee v = \top) \Rightarrow \mathsf{S}_{pc+1}(s) \mid code[pc] = \mathsf{JUMPif}\ i\}\\
&\cup \{\mathsf{S}_{pc}(v :: s) \wedge v \neq 0 \Rightarrow \mathsf{S}_{i}(s) \mid code[pc] = \mathsf{JUMPif}\ i\}
\end{aligned}
$$

This is the last page.