Formal Methods for Security and Privacy (SS 2021) Univ-Prof. Dr. Matteo Maffei

Final Exam

Date: 08/07/2021 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 on : `https://tuwien.zoom.us/j/94784132499?pwd=SEI2ZnlOeWZQeTVFd1lYMWE3ekNUdz09` (passcode WVFyC7Fk).
- **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	4	**Total**
Points	25	25	20	30	100

Problem 1: Typing for Cryptographic Protocols (25 Points)

$$
\begin{array}{rl}
\ell_C, \ell_I & ::= H \mid L \\
\ell & ::= \ell_C \ell_I \\
T & ::= \ell \mid \mathrm{C}^{\ell}[\tilde{T}] \mid \mathrm{SymK}^{\ell}[\tilde{T}]
\end{array}
$$

$$
\begin{array}{rl}
L & \sqsubseteq_C H \\
H & \sqsubseteq_I L \\
\ell_C^1 \ell_I^1 & \sqsubseteq \ell_C^2 \ell_I^2 \iff \ell_C^1 \sqsubseteq_C \ell_C^2 \wedge \ell_I^1 \sqsubseteq_I \ell_I^2 \\
\ell_C^1 \sqcup \ell_C^2 & = \begin{cases} L & \text{if } \ell_C^1 = \ell_C^2 = L \\ H & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{rl}
\ell_1 & \leq \ell_2 \text{ whenever } \ell_1 \sqsubseteq \ell_2 \\
LL & \leq \mathrm{C}^{LL}[LL, \cdots, LL] \\
\mathrm{C}^{\ell}[\tilde{T}] & \leq \ell \\
LL & \leq \mathrm{SymK}^{LL}[LL, \cdots, LL] \\
LL & \leq \mathrm{SigK}^{LL}[LL, \cdots, LL] \\
\mathrm{SymK}^{\ell}[\tilde{T}] & \leq \ell \\
\mathrm{SigK}^{\ell}[\tilde{T}] & \leq \ell
\end{array}
$$

$$
\begin{array}{rl}
\mathcal{L}(\ell) & = \ell \\
\mathcal{L}(\mathrm{C}^{\ell}[\tilde{T}]) & = \ell \\
\mathcal{L}(\mathrm{SymK}^{\ell}[\tilde{T}]) & = \ell \\
\mathcal{L}_{I,\Gamma}(\{|M|\}_K^s) & = \begin{cases} H & \text{if } \Gamma(K) = \mathrm{SymK}^{HH}[T] \\ & \quad \wedge \mathcal{L}_{i,\Gamma}(m) \sqsubseteq_i \mathcal{L}_i(T) \\ L & \text{otherwise} \end{cases} \\
\mathcal{L}_C(\ell_C \ell_I) & = \ell_C \\
\mathcal{L}_I(\ell_C \ell_I) & = \ell_I \\
\mathcal{L}_{\Gamma}(M) & = \begin{cases} \mathcal{L}(\Gamma(M)) & \text{if } M \in dom(\Gamma) \\ LL & \text{otherwise} \end{cases}
\end{array}
$$

$$\textsc{Empty}\ \frac{}{\emptyset \vdash \diamond} \qquad \textsc{Env}\ \frac{\Gamma \vdash \diamond \qquad M \notin dom(\Gamma) \qquad T \in \{\mathrm{C}^{\ell}[\tilde{T}, \mathrm{SymK}^{\ell}[\tilde{T}]], \mathrm{SigK}^{\ell}[\tilde{T}]\} \text{ implies } \ell = HH}{\Gamma, M : T \vdash \diamond}$$

$$\textsc{Atom}\ \frac{\Gamma \vdash \diamond \qquad M : T \text{ in } \Gamma}{\Gamma \vdash M : T} \qquad \textsc{List}\ \frac{\Gamma \vdash M_1 : T_1 \qquad \ldots \qquad \Gamma \vdash M_n : T_n}{\Gamma \vdash [M_1, \ldots, M_n] : [T_1, \ldots, T_n]}$$

$$\textsc{Subsumption}\ \frac{\Gamma \vdash M : T' \qquad T' \leq T}{\Gamma \vdash M : T} \qquad \textsc{SymEnc}\ \frac{\Gamma \vdash K : \mathrm{SymK}^{\ell_C \ell_I}[\tilde{T}] \qquad \Gamma \vdash \tilde{M} : \tilde{T}}{\Gamma \vdash \{|\tilde{M}|\}_K^s : L\ell_I}$$

$$\textsc{VerKey}\ \frac{\Gamma \vdash K : \mathrm{SigK}^{\ell_C \ell_I}[\tilde{T}]}{\Gamma \vdash \mathrm{vk}(K) : \mathrm{VerK}^{L\ell_I}[\tilde{T}]} \qquad \textsc{DigSig}\ \frac{\Gamma \vdash K : \mathrm{SigK}^{\ell_C \ell_I}[\tilde{T}] \qquad \Gamma \vdash \tilde{M} : \tilde{T} \qquad \ell'_C = \sqcup_{T \in \tilde{T}} \mathcal{L}_c(T)}{\Gamma \vdash [\tilde{M}]_K : \ell'_C \ell_I}$$

$$\textsc{Stop}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0}} \qquad \textsc{Par}\ \frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \qquad \textsc{Repl}\ \frac{\Gamma \vdash P}{\Gamma \vdash !P} \qquad \textsc{Res}\ \frac{\Gamma, a : T \vdash P}{\Gamma \vdash (\nu a : T)\ P}$$

$$\textsc{Cond}\ \frac{\Gamma \vdash M : T \qquad \Gamma \vdash N : T' \qquad \Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash \text{if } M = N \text{ then } P \text{ else } Q} \qquad \textsc{In}\ \frac{\Gamma, \tilde{x} : \tilde{T} \vdash P \qquad \Gamma \vdash N : \mathrm{C}^{\ell}[\tilde{T}]}{\Gamma \vdash N(\tilde{x}).P}$$

$$\textsc{Out}\ \frac{\Gamma \vdash \tilde{M} : \tilde{T} \qquad \Gamma \vdash P \qquad \Gamma \vdash N : \mathrm{C}^{\ell}[\tilde{T}]}{\Gamma \vdash \overline{N}\langle \tilde{M} \rangle.P}$$

$$\textsc{SymDec}\ \frac{\Gamma \vdash M : T \qquad \Gamma \vdash K : \mathrm{SymK}^{\ell}[\tilde{T}] \qquad \Gamma, \tilde{x} : \tilde{T} \vdash P}{\Gamma \vdash \text{ case } M \text{ of } \{|\tilde{x}|\}_K^s \text{ in } P}$$

$$\textsc{SignCheck}\ \frac{\Gamma \vdash M : T \qquad \Gamma \vdash K : \mathrm{VerK}^{\ell_C, \ell_I}[\tilde{T}] \qquad \Gamma, \tilde{x} : \tilde{T} \vdash P \qquad \mathcal{L}_C(T) = H \Rightarrow \ell_i = H}{\Gamma \vdash \text{ case } M \text{ of } [\tilde{x}]_K \text{ in } P}$$

Figure 1: Type system for cryptographic protocols

Consider the type system presented in Figure 1, which is a simplified version of the type system presented in the lecture (here we only consider symmetric encryption and digital signatures).

Definition 1 (Opponent). *A process O is an opponent if any $(\nu a : T)$ ocurring in O are such that $T = LL$.*

Definition 2 (Secrecy). *P preserves secrecy if, for all opponents O, whenever $P \mid O \rightarrow^* (\nu c : T)\ (\nu \tilde{a} : \tilde{T})\ (P' \mid \overline{b}\langle c\rangle.P'')$ we have $\mathcal{L}_C(T) \sqsubseteq_C \mathcal{L}_{C,\Gamma}(b)$, with $\Gamma = c : T, \tilde{a} : \tilde{T}$.*

Definition 3 (Integrity). *P preserves integrity if, for all opponents O, whenever*

$$P \mid O \rightarrow^* (\nu b : C^{HH}[T])\ (\nu \tilde{a} : \tilde{T})\ (P' \mid \overline{b}\langle c\rangle.P'') \text{ or}$$
$$P \mid O \rightarrow^* (\nu k : SymK^{HH}[T])\ (\nu \tilde{a} : \tilde{T})\ (P' \mid \text{case } \{|c|\}^s_k \text{ of } \{|x|\}^s_k \text{ in } P'')$$

we have $\mathcal{L}_{I,\Gamma}(c) \sqsubseteq_C \mathcal{L}_I(T)$, with $\Gamma = b : C^{HH}[T], k : SymK^{HH}[T], \tilde{a} : \tilde{T}$.

Theorem 1 (Typing implies Secrecy and Integrity). *Let $\Gamma \vdash P$ with $img(\Gamma) = LL$. Then P preserves both secrecy and integrity.*

Figure 2: Definitions and theorems

a) **(18 Points)** Consider the following process:

$$(\nu k : \mathrm{SigK}^{HH}[LH, LL])\ ((\nu m : LH)\ (\nu n : LH)\ \overline{b}\langle[[m, n]]_k\rangle.\mathbf{0} \mid b(x).\text{case } x \text{ of } [[y, z]]_{\mathrm{vk}(k)} \text{ in } \overline{b}\langle[y, z]\rangle.\mathbf{0})$$

Prove that this process preserves secrecy and integrity, using the theorem from Figure 2.

b) **(7 Points)** In the process of the previous exercise, the signature does not really serve a purpose, i.e., the protocol would also type-check if the signature would be dropped:

$$(\nu k : \mathrm{SigK}^{HH}[LH, LL])\ ((\nu m : LH)\ (\nu n : LH)\ \overline{b}\langle[m, n]\rangle.\mathbf{0} \mid b([y, z]) \text{ in } \overline{b}\langle[y, z]\rangle.\mathbf{0})$$

Give an example of a process that preserves secrecy and integrity, where removing the signature would result in a typing error. Intuitively explain you answer, you do not need to give a typing derivation.

Problem 2: Bana-Comon Logic (25 points)

Predicates

$t \sqsubseteq t_1, \cdots t_n$	iff t is a subterm of some t_i
$t \sqsubseteq_{\not d} t_1, \cdots t_n$	iff any occurrence of t in any t_i is in a decryption key position (i.e., $dec(\cdot, t)$)
$RandGen(t)$	iff $t \in \mathcal{N}$
$Fresh(x; \vec{x})$	iff $RandGen(x) \wedge x \not\sqsubseteq \vec{x}$

Axioms

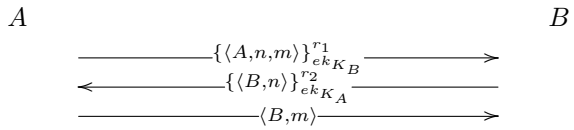
Self derivability	$\vec{x}, x \vdash x$
Increasing capabilities	$\vec{x} \vdash y \rightarrow \vec{x}, x \vdash y$
Commutativity	$\vec{x} \vdash y \rightarrow \vec{x'} \vdash y$ if $\vec{x'}$ is a permutation of $\vec{x}$
Transitivity	$\vec{x} \vdash y \wedge \vec{x}, y \vdash z \rightarrow \vec{x} \vdash z$
Function derivability	$\vec{x} \vdash f(\vec{x})$
No telepathy	$Fresh(x; \vec{x}) \rightarrow \vec{x} \not\vdash x$
Secrecy	$RandGen(K) \wedge ek_K \sqsubseteq \phi \wedge Fresh(R; \phi, x, y) \wedge \phi, \{x\}^R_{ek_K} \vdash y$ $\rightarrow dk_K \sqsubseteq_{\not d} \phi, x \vee \phi \vdash y$

Equations

$$
\begin{aligned}
dec(\{x\}^R_{ek_K}, dk_K) &= x \\
\pi_1(\langle x, y\rangle) &= x \\
\pi_2(\langle x, y\rangle) &= y \\
\pi_1(\langle x, y, z\rangle) &= x \\
\pi_2(\langle x, y, z\rangle) &= y \\
\pi_3(\langle x, y, z\rangle) &= z
\end{aligned}
$$

Figure 3: Rules for the Bana-Comon Logic

Consider the following protocol:



Here $k_A, n, m, r_1 \in \mathcal{N}$ are names known only to A and $k_B, r_2 \in \mathcal{N}$ are names only known to B. A, B are public constants, and ek_{K_A} and ek_{K_B} are public.

a) **(10 points)** Model the protocol as a transition system. Each party should check the correctness of incoming messages as precisely as possible with its knowledge. You can use the predicate $ID(\cdot)$ to check that some term is a valid identifier and you can assume that all names are of length η.

b) **(5 + 10 points)** Let ϕ be the frame that results from the execution of the first step (A sending the initial message to B) from the initial state. Prove that the following statements are inconsistent with the axioms:

(i) $\phi \not\vdash \langle A, B\rangle$

(ii) $\phi \vdash n$

Justify each of your steps using the rules from Figure 3.

Problem 3: Information Flow (20 points)

Rules for typing expressions

$$\textsc{Int}\ \frac{}{n : L} \qquad \textsc{R-Val}\ \frac{\Gamma(x) = \tau var}{\Gamma \vdash x : \tau} \qquad \textsc{Op}\ \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \mathbf{op} \in \{+, -, >, <\}}{\Gamma \vdash e_1\ \mathbf{op}\ e_2 : \tau}$$

Rules for typing commands

$$\textsc{Skip}\ \frac{}{\Gamma \vdash \text{skip} : H\,cmd} \qquad \textsc{Assign}\ \frac{\Gamma(x) = \tau var \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \tau\,cmd}$$

$$\textsc{If}\ \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash c_1 : \tau\,cmd \qquad \Gamma \vdash c_2 : \tau\,cmd}{\Gamma \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 : \tau\,cmd} \qquad \textsc{While}\ \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash c : \tau\,cmd}{\Gamma \vdash \text{while } e \text{ do } c : \tau\,cmd}$$

$$\textsc{Compose}\ \frac{\Gamma \vdash c_1 : \tau\,cmd \qquad \Gamma \vdash c_2 : \tau\,cmd}{\Gamma \vdash c_1; c_2 : \tau\,cmd}$$

Subtyping rules

$$\textsc{Base}\ \frac{}{L \subseteq H} \qquad \textsc{Cmd}^{-}\ \frac{\tau' \subseteq \tau}{\tau\,cmd \subseteq \tau'\,cmd} \qquad \textsc{Reflex}\ \frac{}{\rho \subseteq \rho} \qquad \textsc{Trans}\ \frac{\rho_1 \subseteq \rho_2 \qquad \rho_2 \subseteq \rho_3}{\rho_1 \subseteq \rho_3}$$

$$\textsc{Subsump}\ \frac{\Gamma \vdash p : \rho_1 \qquad \rho_1 \subseteq \rho_2}{\Gamma \vdash p : \rho_2}$$

Figure 4: Type System for Non-interference.

The typing derivation rules presented during the course and that we recall above do not take non-termination into account. For termination-sensitive information flow, the If and While rules have to be more restrictive, namely:

$$\textsc{If}\ \frac{\Gamma \vdash e : L \qquad \Gamma \vdash c_1 : \tau\,cmd \qquad \Gamma \vdash c_2 : \tau\,cmd}{\Gamma \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 : \tau\,cmd} \qquad \textsc{While}\ \frac{\Gamma \vdash e : L \qquad \Gamma \vdash c : \tau\,cmd}{\Gamma \vdash \text{while } e \text{ do } c : \tau\,cmd}$$

Figure 5: Termination-sensitive Type System for Non-interference.

We also need a different characterisation of what it means for a program not to leak, which takes non-termination into account:

Definition 4 (Termination-sensitive Non-interference)**.** *A program p is termination-sensitive non-interferent (i.e., does not leak information, not even through its termination behaviour) if, for all $\mu \sim_L \nu$, whenever executing p in μ terminates and results in μ', then executing p in ν also terminates and results in a state ν' for which $\mu \sim_L \nu$.*

a) **(10 Points)** Find to example, one using the If construction, one using the WHILE construction, such that:

   i) both are well-typed according to the type system in Figure 4 (show a type derivation);
   ii) both are **not** well-typed when replacing the If and While rules by the ones in Figure 5 (point out where and why type-checking necessarily fails);
   iii) one is termination-sensitive non-interferent according to Definition 4, the other is not (explain why and what this implies).

Theorem 2 (Soundness)**.** *If p is well-typed using termination-sensitive versions of* If *and* While *rules given in Figure 5, then p is termination-sensitive non-interferent, i.e., does not leak information from higher to lower levels, even through its termination behaviour.*

b) **(10 Points)** Argue convincingly that Theorem 2 holds (– the closer to an actual proof, the better). You can assume that the type system in Figure 4 is sound w.r.t (non termination-sensitive) non-interference.

Problem 4: Static Analysis (30 points)

Assume the tiny stack-based language with the following instructions:

$$\mathsf{CONST}\ v,\ \mathsf{ADD},\ \mathsf{SUB},\ \mathsf{JUMPifn}\ i \qquad\qquad i \in \mathbb{N}, v \in \mathbb{Z}$$

You can think of programs in this language as simple imperative programs manipulating an implicit stack. Informally, the instruction CONST v put the value v on the top of the stack, the instruction DUP duplicates the value on the top of the stack, the instruction ADD (resp. SUB) adds (resp. substracts) the two values on the top of the stack, and the instruction JUMPifn i jumps to the instruction at program counter i if the value on the top of the stack is negative.

More formally, we describe the state of a program by a configuration of the form $\{pc, s\}$, where pc is the current program counter and s the current stack. $x :: y :: s$ denotes the stacks whose first element is x, whose second element is y, and where s is the rest of the stack. Next, we can define the semantics of our stack-based language in terms of a small-step relation, in which $code \vdash \{pc, s\} \to \{pc', s'\}$ means that, for given a program $code$ (which is just a list of instructions), the configuration changes from $\{pc, s\}$ to $\{pc', s'\}$ within processing one instruction:

$$\text{CONST}\ \frac{code[pc] = \mathsf{CONST}\ v}{code \vdash \{pc, s\} \to \{pc+1, v :: s\}} \qquad \text{DUP}\ \frac{code[pc] = \mathsf{DUP}}{code \vdash \{pc, v :: s\} \to \{pc+1, v :: v :: s\}}$$

$$\text{ADD}\ \frac{code[pc] = \mathsf{ADD}}{code \vdash \{pc, x :: y :: s\} \to \{pc+1, (x+y) :: s\}} \qquad \text{SUB}\ \frac{code[pc] = \mathsf{SUB}}{code \vdash \{pc, x :: y :: s\} \to \{pc+1, (x-y) :: s\}}$$

$$\text{JUMPIFN-TRUE}\ \frac{code[pc] = \mathsf{JUMPifn}\ i \qquad j < 0}{code \vdash \{pc, j :: s\} \to \{i, s\}} \qquad \text{JUMPIFN-FALSE}\ \frac{code[pc] = \mathsf{JUMPifn}\ i \qquad j \geq 0}{code \vdash \{pc, j :: s\} \to \{pc+1, s\}}$$

Figure 6: Concrete Semantics

During the lecture we show how to define an abstract semantics using a very simple abstract domain $D \triangleq \mathbb{Z} \cup \{\top\}$, where $\top$ represents all possible values. In the following we will define an abstract semantics based on the abstract domain of (simple) intervals.

We define the interval abstract domain as $I \triangleq \{[a, b] \,|\, a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}, a \leq b\}$. The abstract value $[a, b] \in I$ is an abstraction of $v \in \mathbb{Z}$ if $a \leq v \leq b$.

a) **(4 Points)** What is the equivalent of $\top$ in I, i.e. what is the abstract value in I which represents all possible values ?

We now want to turn the concrete semantics in Figure 6 into an abstract semantics which involves abstract values of I instead of concrete value of $\mathbb{Z}$. As for static analysis that were presented in the lecture, we define in the following our abstract semantics using Horn clauses. $S_{pc}(s)$ denotes the state predicate which models the abstract configuration at program counter pc over a stack s of abstract values. In Figure 7 we give the rules for CONST, DUP and ADD.

$$\begin{array}{ll} \forall pc \text{ s.t. } code[pc] = \mathsf{CONST}\ v & S_{pc}(s) \Rightarrow S_{pc+1}([v, v] :: s) \\ \forall pc \text{ s.t. } code[pc] = \mathsf{DUP} & S_{pc}([a, b] :: s) \Rightarrow S_{pc+1}([a, b] :: [a, b] :: s) \\ \forall pc \text{ s.t. } code[pc] = \mathsf{ADD} & S_{pc}([a_x, b_x] :: [a_y, b_y] :: s) \Rightarrow S_{pc+1}([a_x + a_y, b_x + b_y] :: s) \end{array}$$

Figure 7: Horn Clauses for CONST, DUP and ADD

$$\forall pc \text{ s.t. } code[pc] = \mathsf{SUB} \quad S_{pc}([a_x, b_x] :: [a_y, b_y] :: s) \Rightarrow S_{pc+1}([a_x - a_y, b_x - b_y] :: s)$$

Figure 8: Unsound Horn Clause for SUB

b) **(6 Points)** The rule for SUB in Figure 8 is unsound. Explain why, give an example and propose a sound version as precise as possible.

c) **(6 Points)** The rules for JUMPifn in Figure 9 are unfinished. Complete them in a sound way and justify your answer.

$$\forall pc \text{ s.t. } code[pc] = \mathsf{JUMPifn}\ i \quad S_{pc}([a,b] :: s) \wedge ??? \Rightarrow S_i(s)$$
$$\forall pc \text{ s.t. } code[pc] = \mathsf{JUMPifn}\ i \quad S_{pc}([a,b] :: s) \wedge ??? \Rightarrow S_{pc+1}(s)$$

Figure 9: Unfinished Horn Clauses for JUMPifn

d) **(8+6 Points)** Using the given rules for CONST, DUP and ADD, the corrected rule for SUB, and the completed rules for JUMPifn:

i) Starting from configuration $S_0([x, y] :: \dots)$ where x and y are fresh variables, what are the abstract states reachable by the following program ? Name the rules (in the right order) that need to be applied to reach the corresponding configurations and give all intermediate configurations.

ii) Use the returned abstract configuration to determine a sufficient condition about the initial configuration which ensures that the concrete evaluation of the program will result in a stack whose top element is strictly positive. If you cannot determine such condition, explain why and propose an alternative.

```
1 CONST 2
2 SUB
3 DUP
4 JUMPifn 7
5 CONST 0
6 SUB
7 CONST 4
8 ADD
```

This is the last page.