Probabilistic Semantic Noninterference

February 27, 2018

1 General Message Passing

1.1 Parties and Schedulers

Assume an expression language with values in Val. A *state* St is a map $\mathsf{Var} \to (\mathsf{Val} + \bot)$, where Var is a set of variable names. A *buffer* is a value of type $\mathsf{list}(\mathsf{PID} \times \mathsf{Val})$, where PID is a set of party names.

A *handler* is given by the following syntax:

$$P := x \leftarrow \mathsf{read}\ \ell \mid \mathsf{write}\ \ell\ e \mid \mathsf{send}\ i\ e \mid x \leftarrow \mathsf{recv}\ i \mid x \leftarrow D\ e \mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P \mid P; P,$$

where e is an expression, ℓ is in Var, i is a PID, and D is a set of distributions.

Handlers are given monadic semantics as functions $\mathsf{St} \to \mathsf{InBuf} \to \mathcal{D}(\mathsf{St} \times \mathsf{OutBuf})$, where InBuf and OutBuf are both buffers. Messages in the input buffer are pairs (i, m), where m is the message content and i is the PID the message came from. Messages in the output buffer, dually, are pairs (i, m), where i is the PID the message is meant for. This semantics is immediate, except for recv, which returns the list of all messages from i in the input buffer (in reverse chronological order).

A *scheduler* is defined by the following syntax:

$$c := \mathsf{Run}\ P\ @\ i \mid c_1;\ c_2,$$

where $k \in \mathsf{Handler}$ is a handler name, and i is a PID.

1.2 Semantics

A *trace* τ is a value of type $(\mathsf{PID} \to \mathsf{St}) \times \mathsf{NewEv} \times \mathsf{OldEv}$, where NewEv and OldEv are logs of the form $\mathsf{list}(\mathsf{PID} \times \mathsf{PID} \times \mathsf{Val})$. The first component is the global state, the second component is ordered buffer of unprocessed messages, and the third component is the ordered buffer of processed messages. Given a buffer B, define $B_{|i}$ to be the pairs in B such that the second component is equal to i (preserving order).

Then, define our scheduler semantics $[\![c]\!] : \mathsf{Trace} \to \mathcal{D}(\mathsf{Trace})$ by

$$[\![\mathsf{Run}\ P\ @\ i]\!](G, B_u, B_p) := \mathsf{bind}_{\mathcal{D}}\ (P\ (G\ i)\ B_{u_{|i}})\ (\lambda(s', o).\ \mathsf{return}_{\mathcal{D}}(G[i := s'], o \mid\mid (B_u \backslash B_{u_{|i}}), B_{u_{|i}} \mid\mid B_p)$$

and

$$[\![c_1;\ c_2]\!]\ \tau\ := \mathsf{bind}_{\mathcal{D}}\ ([\![c_1]\!]\tau)[\![c_2]\!].$$

where $\mathsf{bind}_{\mathcal{D}}$ and $\mathsf{return}_{\mathcal{D}}$ are the monadic bind and return operations for distributions and $||$ is list concatenation.

1.3 Corruption

Extend the syntax of schedulers as so:

$$c := \cdots \mid \mathsf{Corrupt}\ P\ @\ i.$$

The semantics of the added command is exactly the same as that of Run.

We define (static) corruption by the following rewrite rules, given by the judgement $c \vdash_{\mathcal{A}} c'$:

$$\frac{}{c \vdash_{\mathcal{A}} c} \quad \frac{}{\mathsf{Run}\ P\ @\ i \vdash_{\mathcal{A}} \mathsf{Corrupt}\ Q\ @\ i} \quad \frac{c_1 \vdash_{\mathcal{A}} c_1' \quad c_2 \vdash_{\mathcal{A}} c_2'}{c_1; c_2 \vdash_{\mathcal{A}} c_1'; c_2'} \quad \frac{c \vdash_{\mathcal{A}} c'}{c \vdash_{\mathcal{A}} c'; \mathsf{Corrupt}\ Q\ @\ i}$$

Then, define $\mathsf{crupt}(c)$ to be the set of parties $i \in \mathsf{PID}$ such that $\mathsf{Corrupt}\ Q\ @\ i$ appears in c for some Q.

(Note that adversarial programs do not share local state. However, they may freely pass messages to and from each other.)

1.3.1 Other adversarial models

Above, $\vdash_{\mathcal{A}}$ models full malicious corruption. We may recover semihonest corruption (i.e., ordinary party-level noninterference without byzantine faults) by replacing $\vdash_{\mathcal{A}}$ with the weakened rewrite rule:

$$\frac{}{c \vdash_{\mathcal{S}} c} \quad \frac{}{\mathsf{Run}\ P\ @\ i \vdash_{\mathcal{S}} \mathsf{Corrupt}\ P\ @\ i} \quad \frac{c_1 \vdash_{\mathcal{S}} c_1' \quad c_2 \vdash_{\mathcal{A}} c_2'}{c_1; c_2 \vdash_{\mathcal{S}} c_1'; c_2'}$$

In the above rule, we do not change the semantics of any party, but only mark certain parties for corruption. By marking only a single party for corruption, the above definition collapses to ordinary noninterference.

Additionally, we may restrict our corruption model with one that cannot corrupt any party at will, but only a certain subset of the parties. An example of this is *honest-verifier zero knowledge*, where the prover is permitted to be malicious but the verifier is not.

1.4 Noninterference

A *leakage* (or *declassification*) property φ is a function $\mathsf{PID} \to (\mathsf{PID} \to \mathsf{St}) \to \mathsf{Val}$. Given an initial global state G, $\varphi\ i\ G$ denotes the information i should be able to learn from G after the execution of the protocol. Given a set T of PIDs, define $\varphi\ T\ G := \{(i, \varphi\ i\ G) \mid i \in T\}$.

Given two distributions D on traces, write $D \equiv_i D'$ if the marginals $\mathcal{D}(\lambda\ G\ B_u\ B_p.\ (G\ i, B_{p_{|i}}))$ are identical for both D and D'. That is, $D \equiv_i D'$ if from party i's position, D contains exactly the same information as D' on both states and processed messages (including order of messages). Similarly lift up to sets of parties by defining $D \equiv_T D' := \wedge_{i \in T} D \equiv_i D'$.

Fix a corruption model $\vdash$. Given global states G and G', define $G =_T G'$ to be $\forall i \in T, (G\ i) = (G'\ i)$. Then, say that c is *φ-noninterferent* if for all c' such that $c \vdash c'$ and global states G, G',

$$G =_{\mathsf{crupt}(c')} G' \wedge \varphi\, \mathsf{crupt}(c')\, G = \varphi\, \mathsf{crupt}(c')\, G' \implies [\![c']\!]\, (G, \emptyset, \emptyset) \equiv_{\mathsf{crupt}(c')} [\![c']\!]\, (G', \emptyset, \emptyset).$$

That is, c' is φ-noninterferent if whenever two initial global states look identical to the adversary and agree on values of φ, then their induced final traces will appear identical to the adversary.

Note that in the above definition, equivalence of traces is sensitive to order of message delivery – but is only sensitive to message ordering from the perspective of individual parties. That is, two send commands in a handler may be safely reordered if they are sent to different recipients.

1.5 Authenticity

Let θ be a property of traces $\mathsf{PID} \to ((\mathsf{PID} \to \mathsf{St}) \times \mathsf{NewEv} \times \mathsf{OldEv}) \to \{0, 1\}$, and let ϕ be a property of memories $\mathsf{PID} \to (\mathsf{PID} \to \mathsf{St}) \to \{0, 1\}$. Lift ϕ and θ to operate on sets of PIDs by defining $\phi\ T\ G := \wedge_{i \in T} \phi\ i\ G$, and similarly for θ.

Fix a corruption model $\vdash$. Then c is (ϵ, θ, ϕ)-*authentic* if for all c' such that $c \vdash c'$ and G,

$$\Pr_{\tau \leftarrow ([\![c']\!]\ (G, \emptyset, \emptyset))} [\theta\ \mathsf{crupt}(c')\ \tau] > \epsilon \implies \phi\ \mathsf{crupt}(c')\ G.$$

That is, c is (ϵ, θ, ϕ)-*authentic* if whenever the adversary triggers the event θ with probability larger than ϵ, then ϕ must be true of the adversary's initial state.

1.6 Examples

In *multiparty computation*, each party is given an input x_i, and a protocol is devised so that each party receives the value $f(\vec{x})$, but no further information is shared. This is modeled by the leakage function $\phi\, i\, G := f((G\, 1).in, \ldots, (G\, n).in)$.

Functions may also be asymmetric, in which the leakage function is $\phi\ i\ G := f_i((G\ 1).in, \ldots, (G\ n).in)$. A canonical example is *oblivious transfer*, where the sender has two messages m_0 and m_1, and the receiver has a bit b. The sender should learn nothing (i.e., $f_S(m_0, m_1, b) = ()$), while the receiver should learn the bth message (i.e., $f_R(m_0, m_1, b) :=$ if b then m_0 else m_1.)

We may define *zero-knowledge* proofs to be authentic relative to the predicates "the verifier output 1" and "the prover has a correct witness w to the NP-statement x", and noninterferent relative to the leakage function "$R(x, w) = 1$" for the verifier, and no leakage for the prover.

2 Sessioned Parties

Here, we will consider parties and adversaries which are assumed to follow the protocol's intended session structure.

Let St be as before. The type $\mathsf{Party}\ M\ M'$ is defined to be functions $\mathsf{St} \to M \to \mathcal{D}(\mathsf{St} \times M')$. Parties are given by the following syntax:

$$P := x \leftarrow \mathsf{read}\ \ell \mid \mathsf{write}\ \ell\ e \mid \mathsf{send} e \mid x \leftarrow \mathsf{recv}\ \mid x \leftarrow D\ e \mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P \mid P; P,$$

where recv evaluates to the current input message, and send alters the intended output message. (I.e., a second invocation of send will overwrite the first one.) Parties come with a typing relation $\vdash P : \mathsf{Party}\ M\ M'$, which says that all recieves and sends respect M and M'.

Schedulers have the same syntax as before:

$$c := \mathsf{Run}\ P\ @\ i \mid \mathsf{Corrupt}\ P\ @\ i \mid c; c$$

Schedulers are well-typed when they respect the sessions of the parties:

$$\frac{\vdash P : \mathsf{Party}\ M\ M'}{\vdash \mathsf{Run}\ P\ @\ i : \mathsf{Schedule}\ M\ M'} \quad \frac{\vdash P : \mathsf{Party}\ M\ M'}{\vdash \mathsf{Corrupt}\ P\ @\ i : \mathsf{Schedule}\ M\ M'} \quad \frac{\vdash c : \mathsf{Schedule}\ M\ M'' \quad \vdash c' : \mathsf{Schedul}}{\vdash c; c' : \mathsf{Schedule}\ M\ M'}$$