Secure Typed Languages

Andrew Meyers

June 27, 2019

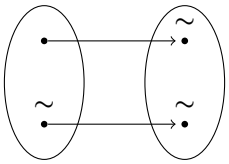
Nondeterministic languages

For nondeterminstic languages, the semantics does not just give one final state. We can define $[\![s]\!]$ to be a set of final states, or a distribution of states, or more exotically, a mixture of distributions of states. The problem is then how to define noninterference $s_1 \sim_L s_2 \implies [\![s_1]\!] \approx_L [\![s_2]\!]$. The question is how to define $\approx_L$ for sets.

[Sutherland '87] defined nondeducibility. The adversary sees a particular final state of the s_1 run, and we ask ourselves whether the result could have been produced by s_2. So we define

$$[\![s_1]\!] \approx [\![s_2]\!] = (\forall t_1 \in [\![s_1]\!] \exists t_2 \in [\![s_2]\!].t_1 \sim_L t_2) \wedge (\forall t_2 \in [\![s_2]\!] \exists t_1 \in [\![s_1]\!].t_2 \sim_L t_1)$$

The following diagram illustrates this relationship.



This property can be enforced with the same type system as we saw last time. We can model nondeterminism with a nondeterministic choice construct $c_1 \Box c_2$. We can typecheck this language with the same rules. For integrity this is good enough but for confidentiality it is not. We give an example.

$$pub := secret \Box put := rand(100)$$

This is secure according to the previous definition of $\approx_L$ if $secret \in 0 \ldots 100$, because for any one run we can't tell that secret has a particular value. Any number that is assigned to *pub* could have been produced by the random number generator. However, if the adversary runs the program many times, it will be able to infer the value of secret because the probability that that value is produced is higher than the probability of the other numbers. Even if the adversary can only run the program once, the adversary still learns something in the information theoretic sense. Note that this is true even if the adversary does not control $\Box$.

The program has an insecure refinement $pub := secret$. This is called a refinement attack. Possibilistic security is not preserved under refinement.

Low-security observational determinism (LSOD)

Roscoe [1] proposed LSOD as a solution. Adversary-observable nondeterminism is subject to refinement attacks. We will require that execution looks deterministic from the point of view of the adversary:

$$[\![s_1]\!] \approx_L [\![s_2]\!] = \forall t_1 \in [\![s_1]\!], t_2 \in [\![s_2]\!].t_1 \sim_L t_2$$

One might think that this property is too restrictive for the programs that we want to write, because the following program is considered insecure:

$$put := true \Box put := false$$

This parallel program is also considered insecure:

$$put := true \parallel pub := false$$

We argue that these programs should be considered insecure. A secret could correlate with the nondeterminism, as in Meltdown and Spectre.

Did we just lose the ability to write parallel programs at all? Consider the following program:

$$(x := 1 \parallel y := 2); \ldots$$

This pattern is called fork-join parallelism. Two independent computations are done in parallel. Such harmless scheduling choices should be allowed. However, consider the following program:

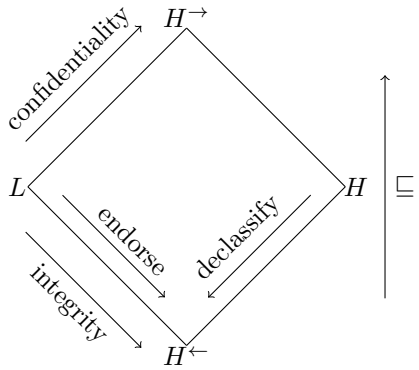
$$(x := true;\ \textbf{if}\ h\ \textbf{then}\ delay(100)\ \textbf{else}\ skip; x := false)\ \parallel (delay(50);\ l := x)$$

This program is insecure, because the race between $x := false$ and $l := x$ allows an adversary to read the value of the secret h using appropriate timing. This is called an internal timing channel. The program above converts the timing channel into a storage channel.

Assume that we have a trace $t = s, s', s'', \ldots$, and assume that locations m are a subset of the locations state. Let $t[m] = s[m], s'[m], s''[m], \ldots$ be the projection of the state on the subset of the locations. We define $t_1 \sim_L t_2$ if the subtraces $t_1[m], t_2[m]$ are indistinguishable at m modulo stuttering (collapse adjacent identical states of the subtraces to a single state). For termination insensitivity, if one trace is finite and the other trace is infinite, we only require them to agree up to the common prefix.

Downgrading

Requiring all information flow to obey the lattice order $\sqsubseteq$ is too restrictive. Real applications need to be able to release secret information. The real security goal is not complete noninterference, but knowledge over how information can flow. We have two dual operations: (1) downgrading confidentiality: declassify(), and (2) downgrading integrity: endorse(). These are represented in the following lattice structure. For those unfamilar with lattice diagrams, it is noted that declassify suggests information flowing $H \sqsubseteq H^{\leftarrow}$, endorse suggests $L \sqsubseteq H^{\leftarrow}$, and that confidentiality increases as $\sqsubseteq H^{\rightarrow}$, and integrity as $\sqsubseteq H^{\leftarrow}$. This suggests that L is potentially public, and to increase confidentiality does not always result in an increase in integrity. It is vital that data with a high confidentiality cannot be modified by data with low integrity, and in general, a secure system will require that no user with integrity less than the confidentiality can modify the data.



The justification for downgrading depends on the application, so we can't expect a type system to check this. We demonstrate this with a few examples.

Example 1: Password Checker

A password check will tell the user whether the password is correct, so the adversary learns something about the secret password:

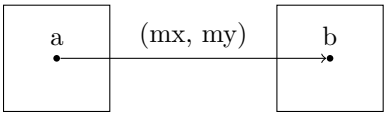
```
if  (declassify(guess == password),H to L)) login = true;
```

Example 2: Battleship

If player A makes a move, we check whether the move was legal, and then B applies the move to their copy of the board:

```
if(legal(mx,my)) endorse(mx,my)(A(<-) to B(<-))
```

The justification for the endorse lies in the rules of the game: A may choose any legal move, and then B must accept that move. In our lattice notation, this means that that if(legal (mx, my)) endorse (mx, my) ($A^{\leftarrow}$ to $B^{\leftarrow}$, then we apply move to B's copy of the board. E.g, (mx, my):$A^{\leftarrow}$. We demonstrate this in the below diagram.



Example 3: average of a set of confidential salaries

We can compute the average of a set of secret salaries and add random noise to make sure that we do not provide too much information about anys individual salary.

```
float arg = declassify(mean(salaries) + noise, H^-> to L)
```

The justification for the declassify comes from differential privacy.
With downgrading we no longer have noninterference:

```
string{H}
bool{L} check(string {H} guess) {
    return declassify(guess == pwd, ...)
}
```

The label H represents confidentiality. The adversary can overwrite the password with a secret, and then test that secret against a guess, thus leaking secret information. This is called laundering secrets via declassify.

This example shows that we need sufficient integrity to securely declassify - we need to ensure that the adversary can't affect what is declassified or whether something is declassified.

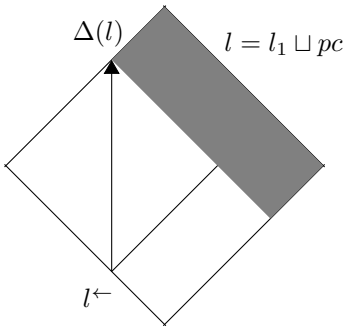
Implementations of this are a message passing concurrent language based on the join calculus [1], Chong et al. [2], JOANA [3]. [2]

Robust declassification

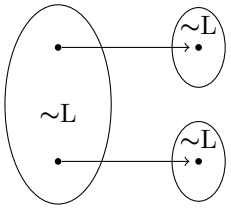
Robust declassification (Zdancewic [3]) is a solution to the problem of secure declassifcation. We define the view operator, which defines the confidentiality level such that the integrity of pc, l are both trusted to declassify.

$$\frac{l_2 \sqsubseteq \Gamma(x) \qquad \Gamma(y) \sqsubseteq l_1 \qquad l_1 \sqsubseteq l_2 \sqcup \bigwedge(l_1 \sqcup pc)}{pc \vdash x := \texttt{declassify}(y, l_1 \texttt{ to } l_2)}$$

The following lattice diagram demonstrates this relationship, as does the mapping of relationships between labels. The shaded region suggests "unsafe" regions and flows, where a secure system should not allow modification or access.



$$\begin{array}{l} H \sqsubseteq L \sqcup H \\ l \sqsubseteq L \sqcup pc \\ L \sqcup H = H \end{array}$$



Robust declassification thus implies that the adversary cannot increase leakage. Leakage will exist, but the adversary cannot increase the amount of information leaked. We define $s[a]$ is defined as the state with an attack a applied to it. a must be a fair attack, it cannot just violate noninterference randomly. Furthermore, it must typecheck within the pc corresponding to L The formal definition of robust declassification semantics is as follows:

$$\forall s, s', a, a'.\ \text{relevant attack } (a) \wedge s[a] \approx_L s'[a] \implies s[a'] \approx_L s'[a']$$

We summarize the number of traces we have to compare to establish each corresponding property in a table.

Method	Traces
Correctness	1 trace
Noninterference, LSOD	2 traces
Robust declassification	4 traces

For those interested in learning more check out the following references:
For a message passing concurrent language see [2]
The X10 programming language implemented some of these ideas. [4]
Java Object-sensitive ANAlysis (JOANA) [5]

References

[1] A William Roscoe. Csp and determinism in security modelling. In *Proceedings 1995 IEEE Symposium on Security and Privacy*, pages 114–127. IEEE, 1995.

[2] Steve Zdancewic and Andrew C Myers. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.*, pages 29–43. IEEE, 2003.

[3] Steve Zdancewic and Andrew C Myers. Robust declassification. 1:15–23, 2001.

[4] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, volume 40, pages 519–538. ACM, 2005.

[5] Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005.