Adversarial Logic

From *Saboteur* to *Arbiter* Models of Incorrectness and Exploitability

Julien Vanegue

IEEE Security & Privacy Langsec Workshop 2023

May 24, 2023

Problem statement

Correctness: Prove the absence of bugs (no false negative)

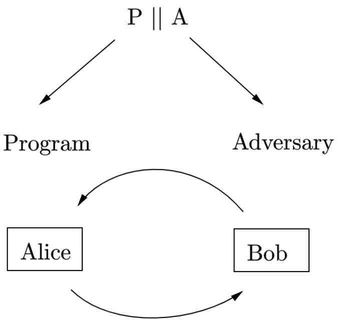
Incorrectness: Prove the presence of bugs (no false positive)

Exploitability: Prove that bugs lead to security vulnerabilities

Exploitability is fundamentally **under-approximate**:
One exploitable path is sufficient to exploit.

Problem: How to distinguish exploitable from non-exploitable bugs?

Adversarial Logic is "Alice and Bob" Program Logic



Motivation: The Oscillating Bit Protocol

```
// pre: client socket established
server(int sock)
{
 uint8 secret = rand8();
 uint8 err = 0;
 uint8 cred = 0;
 while (true) do
   read(sock, cred);
    if (secret == cred)
     err = 0;
    else if (secret < cred)
      err = 1;
    else if (secret > cred)
      err = 2;
    if (!err) do_serve(sock);
    write(sock, err);
 done
}
```

```
// pre: srv socket established
client(int sock)
{
 uint8 ret = 1;
 uint8 guess = UINT8_MAX;
 uint8 step = (UINT8_MAX/2)+1;
 while (true) do
    write(sock, guess);
    read(sock, ret);
    if (ret == 1)
      guess = guess - step;
    else if (ret == 2)
      guess = guess + step;
   step = (step / 2) + 1;
   adv_assert(ret == 0);
 done
}
```

Hoare Logic, Incorrectness Logic, and Adversarial Logic

Hoare Logic: $\{P\}\ c\ \{Q\}$

Incorrectness Logic: $[P]\ c\ [ok\colon Q][er\colon R]$ aka $[P]\ c\ [\epsilon : Q]$

Adversarial Logic (A Parallel Incorrectness Logic):

$\{ok\colon P\}\ c_p\ \{ok\colon Q\}$
$\{ad\colon A\}\ c_a\ \{ad\colon B\}$
$\{\epsilon : X\}\ c\ \{\epsilon : Y\}$
$\{ok\colon P\}\{ad\colon A\}\ p \mid\mid a\ \{ok\colon Q\}\{ad\colon B\}$

with $\epsilon \in \{ok, ad\}$

Adversarial Logic In A Nutshell

- IL with added adversarial transitivity for error composition.
- Program and Adversary are composed in parallel.
- P and A share no variable, exchange messages on channels.
- Adversary decides what assertion must hold for attack success.
- Adversarial assertion only needs to be satisfied on one path.

Variables	V ::= x | n | α
Expressions	E ::= V | rand() | E + E | E - E | ...
Channels	L ::= s | $\emptyset$ | (V::L) | (L::V) | (s $\setminus$ V)
Predicates	B ::= B $\wedge$ B | B $\vee$ B | $\neg$B | E == E | E $\leq$ E | ...
Data types	T ::= uint8 | uint32 | float
Commands	C ::= skip | x := E | s := L | $C_1; C_2$ | $C_1 \parallel C_2$
	| if B then C_1 else C_2
	| while B do C done
	| read(s, x)
	| write(s, E)
	| adv_assert(B)
	| T x = E in C
	| Com(C_1,C_2)

Adversarial Logic Structure

AL has three fragments:

- **Core rules**: The same as incorrectness logic with ϵ symmetry restored.
- **Communication rules**: Used to communicate with messages between programs.
- **Knowledge rules**: Used by the adversary to infer new knowledge.

Adversarial Logic: Core rules with $\epsilon \in \{ok, ad\}$

$$\textbf{Unit}\ \frac{}{[\epsilon: P]\ \mathrm{skip}\ [\epsilon: P]} \qquad \textbf{Constancy}\ \frac{[\epsilon: P]c[\epsilon: Q]}{[\epsilon: P \wedge F]c[\epsilon: Q \wedge F]}\ Mod(c) \cap Free(F) = \emptyset$$

$$\textbf{Consequence}\ \frac{[\epsilon: P \Rightarrow P'] \qquad [\epsilon: P]c[\epsilon: Q] \qquad [\epsilon: Q' \Rightarrow Q]}{[\epsilon: P']c[\epsilon: Q']}$$

$$\textbf{Assume}\ \frac{}{[\epsilon: P]\ \mathrm{assume(B)}\ [\epsilon: P \wedge B]} \qquad \textbf{Rand}\ \frac{}{[\epsilon: P]\ \mathrm{x = rand()}\ [\epsilon: \exists x'.P(x'/x) \wedge x = v]}$$

$$\textbf{Assign}\ \frac{}{[\epsilon: P]\ \mathrm{x = e}\ [\epsilon: \exists x'.P(x'/x) \wedge x = e(x'/x)]}$$

$$\textbf{Disj}\ \frac{[\epsilon: P_1]c[\epsilon: Q_1] \qquad [\epsilon: P_2]c[\epsilon: Q_2]}{[\epsilon: P_1 \vee P_2]c[\epsilon: Q_1 \vee Q_2]} \qquad \textbf{Local}\ \frac{[\epsilon: P \wedge x = e]c[\epsilon: Q] \qquad x \notin Free(P)}{[\epsilon: P]\ \mathrm{T\ x = e\ in\ c}\ [\epsilon: \exists x \in \mathbb{T}.Q]}$$

$$\textbf{Seq}\ \frac{[\epsilon: P]c_1[\epsilon: Q] \qquad [\epsilon: Q]c_2[\epsilon: R]}{[\epsilon: P]c_1; c_2[\epsilon: R]} \qquad \textbf{Choice}\ \frac{[\epsilon: P]c_i[\epsilon: Q]}{[\epsilon: P]c_1 + c_2[\epsilon: Q]}\ i \in [1, 2]$$

$$\textbf{Iterate Zero}\ \frac{}{[\epsilon: P]c^*[\epsilon: P]} \qquad \textbf{Iterate non-zero}\ \frac{[\epsilon: P]c^*; c[\epsilon: Q]}{[\epsilon: P]c^*[\epsilon: Q]}$$

$$\mathrm{while}\ B\ \mathrm{do}\ C\ \mathrm{done} \triangleq (\mathrm{assume}(B);\ C)^*;\ \mathrm{assume}(\neg B)$$

$$\mathrm{if}\ B\ \mathrm{then}\ C\ \mathrm{else}\ C' \triangleq (\mathrm{assume}(B);\ C) + (\mathrm{assume}(\neg B);\ C')$$

Adversarial Logic: Communication rules

$$\textbf{Read}\ \frac{s \in \mathit{Chan}(P)}{[\epsilon\colon P]\ \text{read(s, x)}\ [\epsilon\colon \exists v \exists x' \exists s'. P(s'/s, x'/x) \wedge s = (s' \backslash v) \wedge x = v]}$$

$$\textbf{Write}\ \frac{s \in \mathit{Chan}(P)}{[\epsilon\colon P \wedge x = v]\ \text{write(s, x)}\ [\epsilon\colon \exists s'. P(s'/s) \wedge s = (s' {::} v)]}$$

$$\textbf{Par}\ \frac{[\epsilon_1\colon P_1] c_1 [\epsilon_1\colon Q_1] \qquad [\epsilon_2\colon P_2] c_2 [\epsilon_2\colon Q_2]}{[\epsilon_1\colon P_1][\epsilon_2\colon P_2]\ c_1 \parallel c_2\ [\epsilon_1\colon Q_1][\epsilon_2\colon Q_2]}\ \epsilon_1, \epsilon_2 \in \{ok, ad\}$$

$$\textbf{Com}\ \frac{s \in \mathit{Chan}(P) \cap \mathit{Chan}(A) \qquad \epsilon_1, \epsilon_2 \in \{ok, ad\}}{[\epsilon_1\colon P][\epsilon_2\colon A]\ c_1 \parallel c_2\ [\epsilon_1\colon \exists v \exists s'. P(s'/s) \wedge s = (s' \backslash v)][\epsilon_2\colon \exists v \exists s'. A(s'/s) \wedge s = (s' {::} v)]}$$

Adversarial Logic: Knowledge rules

$$\textbf{PBV}\ \frac{[ok\colon P(n)][ad\colon A(m)]\ c_p \mid\mid c_a\ [ok\colon P(n+i)][ad\colon A(m+j)]}{[ok\colon P(0)][ad\colon A(0)]\ c_p^n \mid\mid c_a^m\ [ok\colon \exists n.P(n)][ad\colon \exists m.A(m)]}\ i,j \in \{0,1\} \wedge i+j \geq 1$$

$$\textbf{AdC}\ \frac{\begin{array}{c}[ok\colon P \wedge s = l_s \wedge v_1 = w]\ c_p\colon \text{if (Q) write(s,}v_1\text{)}\ [ok\colon P' : \exists s'.P(s'/s) \wedge s = (l_s{::}w) \wedge Q] \\ [ad\colon A \wedge s = (w{::}l_a)]\ c_a\colon \text{read(s,}v_2\text{)}\ [ad\colon A' : \exists s', v_2'.A(s'/s, v_2'/v_2) \wedge s = l_a \wedge v_2 = w]\end{array}}{[ok\colon P][ad\colon A]\ c_p \mid\mid c_a\ [ok\colon P'][ad\colon A' \wedge \exists v_1.Q \wedge v_1 = v_2]}$$

$$\textbf{Success}\ \frac{Q \Rightarrow B}{[ad\colon Q]\ \text{adv\_assert(B)}\ [ad\colon Q \wedge true]}$$

$$\textbf{Failure}\ \frac{Q \Rightarrow \neg B}{[ad\colon Q]\ \text{adv\_assert(B)}\ [ad\colon Q \wedge \neg B]}$$

Simplest “CTF” example

`// Precond: s channel established`
`program(int s)`
`{`
`  uint32 n, win in`
`  read(s, n);`
`  if (n > 10M) win = 1;`
`  else win = 0;`
`  write(s, win);`
`}`
	`// Precond: s established`
	`adversary(int s)`
	`{`
	`  uint32 val = α in`
	`  uint32 res = 0 in`
	`  write(s, val);`
	`  read(s, res);`
	`  adv_assert(res == 1);`
	`}`

Simplest "CTF" example in AL

	Program	Adversary	
$\rightarrow$ *Local*	Program(int s) {	Adversary(int s) {	
	$P_0 = \{ok\text{: } \exists s_p . s_p = \emptyset\}$	$A_0 = \{ad\text{: } \exists s_a . s_a = \emptyset\}$	
$\rightarrow$ *Local*	uint32 n in	uint32 val $= \alpha$ in	$\rightarrow$ *Local*
	$P_1 = \{ok\text{: } P_0 \wedge \exists u . n = u\}$	$A_1 = \{ad\text{: } A_0 \wedge \exists \alpha . val = \alpha\}$	
$\rightarrow$ *Local*	uint32 win in	uint32 res $= 0$ in	$\rightarrow$ *Local*
	$P_2 = \{ok\text{: } P_1 \wedge \exists v . win = v\}$	$A_2 = \{ad\text{: } A_1 \wedge res = 0\}$	
$\rightarrow$ *Skip*	read(s, n);	write(s,val);	$\rightarrow$ *Write*
	$P_3 = \{ok\text{: } P_2\}$	$A_3 = \{ad\text{: } \exists s_a^2 . A_2(s_a^2/s_a) \wedge s_a = (s_a^2{::}\alpha)\}$	

$\rightarrow$ *Par* $\quad (P_3, A_3) = \{ok\text{: } P_3\}\{ad\text{: } A_3\}$

$\rightarrow$ *Com* $\quad$ read(s,n) || read(s,res)

$(P_4, A_4) = \{ok\text{: } \exists \alpha \exists s_p^2 . P_3(s_p^2/s_p) \wedge s_p = (s_p^2{::}\alpha)\}\{ad\text{: } \exists s_a^2 . A_3(s_a^2/s_a) \wedge s_a = (s_a^2 \backslash \alpha)\}$

$\rightarrow$ *Read* $\quad$ read(s,n) || read(s, res)

$(P_5, A_5) = \{ok\text{: } \exists \alpha \exists s_p^3 \exists n_2 . P_4(s_p^3/s_p, n_2/n) \wedge s_p = (s_p^3 \backslash \alpha) \wedge n = \alpha\}\{ad\text{: } A_4\}$

$\rightarrow$ *If*, *Assn* $\quad$ if (n $>$ 10M) win = 1 || read(s, res)

$(P_6, A_6) = \{ok\text{: } \exists w_2 . P_5(w_2/win) \wedge n > 10M \wedge win = 1\}\{ad\text{: } A_5\}$

$\rightarrow$ *If*, *Assn* $\quad$ else win = 0 || read(s, res)

$(P_7, A_7) = \{ok\text{: } \exists w_3 . P_5(w_3/win) \wedge n \leq 10M \wedge win = 0\}\{ad\text{: } A_6\}$

$\rightarrow$ *Disj* $\quad (P_8, A_8) = \{ok\text{: } P_6 \vee P_7\}\{ad\text{: } A_7\}$

$\rightarrow$ *Write* $\quad$ write(s, win) || read(s, res)

$(P_9, A_9) = \{ok\text{: } \exists w \exists s_p^4 . P_8(s_p^4/s_p) \wedge s_p = (s_p^4{::}w) \wedge win = w\}\{ad\text{: } A_8\}$

- -

$\rightarrow$ *Com* $\quad$ skip || read(s,res)

$(P_{10}, A_{10}) = \{ok\text{: } \exists s_p^5 . P_9(s_p^5/s_p) \wedge s_p = (s_p^5 \backslash w)\}\{ad\text{: } \exists w \exists s_a^3 . A_9(s_a^3/s_a) \wedge s_a = (s_a^3{::}w)\}$

$\rightarrow$ *Read* $\quad$ skip || read(s, res)

$(P_{11}, A_{11}) = \{ok\text{: } P_{10}\}\{ad\text{: } \exists s_a^4 \exists r_2 \exists w . A_{10}(s_a^4/s_a, r_2/res) \wedge s_a = (s_a^4 \backslash w) \wedge res = w\}$

$\rightarrow$ *Adv.C.* $\quad$ skip || adv_assert(res == 1)

$(P_{12}, A_{12}) = \{ok\text{: } P_{11}\}\{ad\text{: } A_{11} \wedge \exists n \exists x . ((n > 10M \wedge x = 1) \vee (n \leq 10M \wedge x = 0)) \wedge n = \alpha \wedge res = x\}$

$\rightarrow$ *Success* $\quad$ skip || adv_assert(res == 1)

Oscillating Bit Protocol: Bootstrap phrase

	$P_1 = \{ok\colon \emptyset\}$	$A_1 = \{ad\colon \emptyset\}$	
$\rightarrow$ *Rand*	uint8 secret = rand8() in	uint8 ret = 1 in	$\rightarrow$ *Local*
	$P_2 = \{ok\colon \exists s.secret == s\}$	$A_2 = \{ad\colon A_1 \wedge ret = 1\}$	
$\rightarrow$ *Local*	uint8 err = 0 in	uint8 guess = UINT8_MAX in	$\rightarrow$ *Local*
	$P_3 = \{ok\colon P_2 \wedge err = 0\}$	$A_3 = \{ad\colon A_2 \wedge guess = UINT8\_MAX\}$	
$\rightarrow$ *Local*	uint8 cred = 0 in	uint8 step = (guess / 2) + 1 in	$\rightarrow$ *Local*
	$P_4 = \{ok\colon P_3 \wedge cred = 0\}$	$A_4 = \{ad\colon A_3 \wedge step = (guess/2) + 1\}$	
$\rightarrow$ *While*	while (true) do	while (true) do	$\rightarrow$ *While*
	$P_5 = \{ok\colon true \wedge P_4\}$	$A_5 = \{ad\colon true \wedge A_4\}$	
$\rightarrow$ *Skip*	read(sock, cred);	write(sock,guess);	$\rightarrow$ *Write*
	$P_6 = \{ok\colon P_5\}$	$A_6 = \{ad\colon \exists s_a^1 \exists g.A_5(s_a^1/s_a)$ $\wedge guess = g \wedge\ s_a = (s_a^1{::}g)\}$	

$\rightarrow$ *Par* $\quad (P_7, A_7) = \{ok\colon P_6\}\{ad\colon A_6\}$

Oscillating Bit Protocol: Communication phase 1

$\rightarrow$ *Par*	$(P_7, A_7) = \{ok\colon P_6\}\{ad\colon A_6\}$
$\rightarrow$ *Com*	read(sock, cred) || read(sock,ret)
	$(P_8, A_8) = \{ok\colon \exists w \exists s_p^1 . P_7(s_p^1/s_p) \wedge s_p = (s_p^1 \backslash w)\}\{ad\colon \exists w \exists s_a^1 . A_7(s_a^1/s_a) \wedge s_a = (s_a^1{::}w)\}$
$\rightarrow$ *Read*	read(sock, cred) || read(sock,ret)
	$(P_9, A_9) = \{ok\colon \exists c \exists s_p^2 . P_8(s_p^2/s_p, c/cred) \wedge s_p = (s_p^2 \backslash c) \wedge cred = c\}\{ad\colon A_8\}$
$\rightarrow$ *If*, *Disj*	if (secret == cred) err = 0 || read(sock,ret)
	$(P_{10}, A_{10}) = \{ok\colon P_9 \vee (secret = cred \wedge \exists e . P_9(e/err) \wedge err = 0)\}\{ad\colon A_9\}$
$\rightarrow$ *If*, *Disj*	else if (secret < cred) err = 1 || read(sock,ret)
	$(P_{11}, A_{11}) = \{ok\colon P_{10} \vee (secret < cred \wedge \exists e . P_{10}(e/err) \wedge err = 1)\}\{ad\colon A_{10}\}$
$\rightarrow$ *If*, *Frame*	if (err == 0) do_serve(sock) || read(sock,ret)
	$(P_{12}, A_{12}) = \{ok\colon (P_{11} \wedge err \neq 0) \vee (P_{11} \wedge err = 0)\}\{ad\colon A_{11}\}$
$\rightarrow$ *Write*	write(sock, err) || read(sock, ret)
	$(P_{13}, A_{13}) = \{ok\colon \exists e \exists s_p^3 . P_{12}(s_p^3/s_p) \wedge err = e \wedge s_p = (s_p^3{::}e)\}\{ad\colon A_{12}\}$

Oscillating Bit Protocol: Final phase and decision

$\rightarrow$ *Com*	read(sock, cred) || read(sock, ret)
	$(P_{14}, A_{14}) = \{ok\colon \exists w \exists s_p^4 . P_{13}(s_p^4/s_p) \wedge s_p = (s_p^4 \backslash w)\}\{ad\colon \exists w \exists s_a^2 . A_{13}(s_a^2/s_a) \wedge s_a = (s_a^2 :: w)\}$
$\rightarrow$ *Read*	read(sock, cred) || read(sock, ret)
	$(P_{15}, A_{15}) = \{ok\colon P_{14}\}\{ad\colon \exists r \exists r_2 \exists s_a^3 . A_{14}(s_a^3/s_a, r_2/ret) \wedge s_a = (s_a^3 \backslash r) \wedge ret = r\}$
$\rightarrow$ *If , Disj*	read(sock, cred) || if (ret == 1) guess = guess - step
	$(P_{16}, A_{16}) = \{ok\colon P_{15}\}\{ad\colon (ret = 1 \wedge \exists g . A_{15}(g/guess) \wedge guess = g - step) \vee (ret \neq 1 \wedge A_{15})\}$
$\rightarrow$ *If , Disj*	read(sock, cred) || if (ret == 1) guess = guess - step
	$(P_{17}, A_{17}) = \{ok\colon P_{16}\}\{ad\colon (ret = 2 \wedge \exists g . A_{16}(g/guess) \wedge guess = g + step) \vee (ret \neq 2 \wedge A_{16})\}$
$\rightarrow$ *Assign*	read(sock, cred) || step = (step / 2) + 1
	$(P_{18}, A_{18}) = \{ok\colon P_{17}\}\{ad\colon \exists s . A_{17}(s/step) \wedge step = s/2 + 1)\}$
$\rightarrow$ *Fail*	read(sock, cred) || adv_assert(ret == 0)
	$(P_{19}, A_{19}) = \{ok\colon P_{18}\}\{ad\colon A_{18} \wedge ret \neq 0)\}$
$\rightarrow$ *PBV*	$(P_{20}, A_{20}) = \{ok\colon \exists n . P_n : (secret = cred) \wedge (err = 0)\}\{ad\colon \exists n . A_n : (ret = 0)\}$
$\rightarrow$ *Success*	read(sock, cred) || adv_assert(ret == 0)

Relational Analysis in Adversarial Logic

- Relational Logic: compare two or more program executions.
- Executions belong to a single program or two comparable programs.
- Shift from a *Saboteur* to an *Arbiter* model of adversary.
- IL/AL cannot prove simulation results due to under-approximation.

Relational Example: URL parsing in python (urllib vs rfc3986)

```
import urllib3
http = urllib3.PoolManager()
b = urllib3.util.parse_url('evil.com://good.com')
Url(scheme=None, auth=None, host='evil.com', port=None,
    path='//good.com', query=None, fragment=None)
a = urlparse('evil.com://good.com')
ParseResult(scheme='evil.com', userinfo=None, host='good.com')
```

See *dippy_gram: Grammar-Aware, Coverage-Guided Differential Fuzzing* by Ben Kallus and Sean W. Smith (Dartmouth College), Langsec 2023

Parser differential: URL confusion attack

```
parseURL1(uint32 s1) {
  string url, host in
  uint32 i = 0, dot = 0;
  uint32 d = 0, len = 0;
  len = read(s1, url);
  while (i < len) {
   if (url[i] = ':')
     d = i;
   else if (url[i] = '.')
     dot = i;
   i++;
  }
  if (dot ≠ 0 && d ≠ 0 &&
    dot < d)
    host = url[:d];
  else if (dot == 0 && d ≠ 0)
    host = url[d:];
  else
    host = url;
  write(s1, host);
}
```

```
parseURL2(uint32 s2) {
  string url, host;
  uint32 i = 0, d = 0, len = 0;
  len = read(s2, url);
  while (i < len) {
    if (url[i] = ':')
      d = i;
    i++;
  }
  if (d ≠ 0) host = url[:d];
  else host = url;
  write(s2, host);
}
adv(uint32 s1, uint32 s2) {
  string q1, q2;
  write(s1,α)
  read(s1,q1)
  write(s2,α)
  read(s2,q2)
  adv_assert(q1 ≠ q2)
}
```

URL parsing attack in Adversarial Logic

$$P_1 \mid\mid P_2 \mid\mid A$$

References

Incorrectness Logic (Peter O'Hearn, POPL 2020)
Incorrectness Separation Logic (Raad et al. CAV 2021)
Concurrent Incorrectness Separation Logic (Raad et al. POPL 2022)
Adversarial Logic (Julien Vanegue, SAS 2022)
Relational Adversarial Logic (Julien Vanegue, submitted to FMSD)

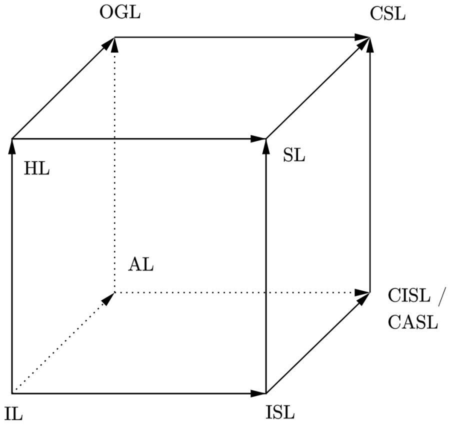
Logic mixing verification and incorrectness reasoning:

Local Completeness Logic (Bruno, Giacobazzi, Gori, Ranzato, 2021)
Outcome Logic (Zilbertein, Dreyer and Silva, 2023)
Hyper-Hoare Logic (Dardinier and Muller, 2023)

Caveats and the future

- AL has *interleaving* semantics. It needs true concurrency.
- AL is unable to reason about pointers (only integers)
  CASL: Concurrent Adversarial Separation Logic
  with Azalea Raad, Peter O'Hearn and Josh Berdine
- AL is unable to reason about non-termination
  INTL: INcorrectness For Termination Logic
  Work in progress with Azalea Raad and Peter O'Hearn

The Program Logic Cube for Correctness And Incorrectness



HL: Hoare Logic
IL: Incorrectness Logic
OGL: Owicki-Gries Logic
SL: Separation Logic
CSL: Concurrent SL
ISL: Incorrectness SL
AL: Adversarial IL
CISL: Concurrent ISL
CASL: Concurrent Adversarial SL

→ Separation ↗ Concurrency ↑ Over-approximation

What are your questions?