Relational Adversarial Logic

Julien Vanegue (julien.vanegue@gmail.com)  
Bloomberg (United States)

Research Article

Keywords: incorrectness, adversarial, logic, security, exploit, under-approximate

Posted Date: May 3rd, 2023

DOI: https://doi.org/10.21203/rs.3.rs-2874730/v1

License: © This work is licensed under a Creative Commons Attribution 4.0 International License. Read Full License

Additional Declarations: No competing interests reported.
Relational Adversarial Logic

Anonymized author

Anonymized Institution

Abstract. Adversarial Logic [1] is an extension of Incorrectness Logic [2] providing a formal framework to study security exploits. This article introduces Relational Adversarial Logic (RAL), a derived logic from AL capable of reasoning on attack proofs for software security hyper-properties. Unlike other approaches to under-approximate relational analysis, we model the relation between two comparable implementations by introducing an explicit adversarial term, which is composed in parallel to the programs under comparison. Unlike in the original Adversarial logic where the parallel composition is used to model the adversary as a saboteur, the adversarial term is used as an arbiter in RAL. As such, RAL can express relational under-approximation without introducing meta-variables or a higher level logic. We demonstrate the relational capabilities of RAL by modeling a real URL confusion vulnerability between two widespread URL parsing implementations. Our presentation demonstrates that it only suffices to introduce new derived rules from AL, rather than introducing a new logic or relying on meta-variables as done in other approaches. As such, RAL retains simplicity and is more faithful to the original incorrectness logic by O’Hearn, from which it inherits its soundness.

Keywords — incorrectness, adversarial, logic, security, exploit, under-approximate

1 Statements and Declarations

The author has no competing interests to declare that are relevant to the content of this article.
2 Introduction

As the number of security vulnerabilities keep growing, many organizations developing large software products find themselves submerged by a deluge of bugs, leading to thousands of issues being known at any time. As new defects are found faster than they are fixed, fixing security issues is a time-consuming expert job in need of automation. Testing and merging fixes in production releases takes weeks or months of work for teams of security engineers to validate.

A more affordable approach to handle such volume of issue is to prioritize bug fixing. Indeed, not all bugs are security vulnerabilities, and the cost of not fixing certain issues for some time is not the same for every bug. Particularly sensitive are these issues leading to untrusted code execution, secret leakage, and other information disclosure weakening the security posture of programs due to revealing or corrupting internal information. Exploiting these security vulnerabilities for malicious purpose lead to costly compromise of cryptographic key material, leak of personal information, and it is widely admitted in the security community that bugs leading to such exploits should be fixed first, especially as such exploitation starts to become more automated [3] [4] [5].

Such a sheer amount of software bugs can come as a surprise: for fifty years, the programming language research community has developed the field of program verification, where techniques such as model checking [6] and abstract interpretation [7] overapproximate the analyzed program behavior to guaranteed the absence of bugs. Such techniques have been applied to a wide variety of programs for a variety of properties, including security properties. For example, the SLAM model checker [8] identified thousands of issues in Windows device drivers, many of them being security vulnerabilities.

A useful realization for bug prioritization is that exploiting a security vulnerability is a fundamentally underapproximate problem. Indeed, it suffices for one particular program path to be exploitable for a bug to constitute a vulnerability. Following a recent trend of underapproximate incorrectness program logic [2], a new adversarial incorrectness logic [1] uses program underapproximation to discover exploitable path. In such adversarial formulation, the attacker (A) is represented as an explicit agent composed in parallel with the target program (P) as \( A \parallel P \) and communicates via messages, guaranteeing that attacker variables and target variables are strictly disjoint.

In this article, we extend adversarial logic (AL) by demonstrating its application to relational analysis of security vulnerabilities. While the adversary is interpreted as a saboteur in the original introduction of AL [1], we show that interpreting the adversarial program as an arbiter allows to encode relational analysis within the logic, and allows to stitch executions of several programs to compare different implementations of the same function without requiring meta-level rules of variables. The same kind of parallel composition is used to encode \( A \parallel P_1 \parallel P_2 \) where \( P_1 \) and \( P_2 \) are two comparable programs under relational analysis, and \( A \) makes use of assertions to encode the relational attack contract. We will show a first practical example of such setup to compare two URL parsing implementations in section 3.

This article is organized as follow: We first give a motivational example in section 3. We then define the rules of RAL and the toy language in which our examples are expressed in section 4. We follow by three examples in which RAL is applied in section 5. The denotational semantics and soundness proof for (R)AL are made explicit in section 6. We conclude by related work in section 7.
3 Motivation

In this section, we will use a real-world motivating example of relational under-approximate adversarial checking. In table 1, we show a minimized example in python where two calls to the `parse_url` function with the same input yields different results. The first implementation of URL parsing in `urllib3` returns the host `evil.com` and scheme `None`. This behavior is due to the legacy RFC implementation which forbid dot characters in schemes. The second implementation from `rfc3986` has no such limitation and will parse the input string differently, leading to a value of scheme of `evil.com` and host value to `good.com`.

Such disagreement between different implementations of the same function is problematic. Not only non-determinism leads to weaknesses in software interoperability, but such interoperability can lead to security issues. For example, a network intrusion detection system examining every URLs by parsing them with the most recent `rfc3986` implementation may think that `good.com` is an acceptable destination, but some clients may use a different implementation following the legacy standard and leading them to visit `evil.com`. Table 2 shows a simplified version of the URL parsing implementation in both libraries.

Table 1: URL parsing confusion in python: default urllib versus rfc3986

```python
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')
```

In the first example shown in `parseURL1` at lines 10-12, the parser will check whether characters `.` or `:` are part of the input string and remember the index of such occurrences if present. As the legacy URL parsing implementation refuses dots in schemes, the input string will be decomposed between a hostname (`evil.com`) and a path (`//good.com`). In the second implementation of `parseURL2`, dot characters are allowed in URL schemes, and therefore no special treatment is given to them. As such, the URL is parsing as a scheme `evil.com` and hostname `good.com`. If no dot is present in the input string, both function returns the exact same values.

In AL, special primitives `read` and `write` are used to model interaction between the adversarial program and the target programs, where messages communicated on channels are used in lieu of parameter passing and return values. This allows AL to model interactive protocols in a more flexible way than forcing parameter passing and value return. We define the full details of the toy language in section 4.

An early implementation explanation is in order to understand the motivational example of table 2. Strings are represented as arrays of characters and expression
Array failsafe $[d:]$ represents the sub-array from index $d$ to the end of the array, while expression $[0:d]$ represents the sub-array from the beginning of the original array until the index $d$. We refer to section 4 for the complete grammar of the input language used in our examples.

Table 2: URL parsing confusion attack: urllib vs. rfc3986. Two mainstream python URL parsing routines return a different result when URL $\alpha = $ "evil.com://good.com"  

```cpp
// pre: channel s_1 established
1. parseURL1(uint32 s_1) {
2. string url in
3. string host in
4. uint32 i = 0 in
5. uint32 dot = 0 in
6. uint32 delim = 0 in
7. uint32 len = 0 in
8. len = read(s_1, url);
9. while (i < len) {
10. if (url[i] = ':')
11. delim = i;
12. else if (url[i] = '.')
13. dot = i;
14. i++;
15. }
16. if (dot ≢ 0 &&
17. delim ≢ 0 &&
18. dot < delim)
19. host = url[:delim];
20. else
21. host = url;
22. write(s_1, host);
23. }

// pre: channel s_2 established
24. parseURL2(uint32 s_2) {
25. string url in
26. string host in
27. uint32 i = 0 in
28. uint32 delim = 0 in
29. uint32 len = 0 in
30. len = read(s_2, url);
31. while (i < len) {
32. if (url[i] = ':')
33. delim = i;
34. i++;
35. }
36. if (delim ≢ 0)
37. host = url[:delim];
38. else
39. host = url;
40. write(s_2, host);
41. }

// pre: channels established
42. adv(uint32 s_1, uint32 s_2) {
43. string q_1, q_2;
44. write(s_1, $\alpha$)
45. read(s_1, $\alpha$)
46. write(s_2, $\alpha$)
47. read(s_2, $\alpha$)
48. adv_assert(q_1 ≢ q_2)
49. }
```

Note also the use of a symbolic variable $\alpha$, which can only be used by the attacker in AL. In a nutshell, we reuse the same symbolic variable for both interactions with parseURL1 and parseURL2 at lines 44-47 so that our checking algorithm knows the input value has to be the same for both calls, and will solve the assertion for such value
of $\alpha$. Assertions are solely provided by the attacker in AL, and only one attack path is sufficient to satisfy the assertion. If the results of both calls are different in result variables $q1$ and $q2$, the attacker has successfully identified a parsing confusion. This is captured by the adversarial assertion on line 48 of table 2. If a value for $\alpha$ exists such that the adversarial assertion is satisfied, the attacker has succeeded. Otherwise, AL makes no claim as to whether such an attack exists or not.

Our motivation to use a pythonic language is two-fold. First, we demonstrate that incorrectness and adversarial logic can be applied to dynamic languages just the same as statically compiled language as done in the original presentations [2] [1]. Secondly, most recent research has focused on extending incorrectness logic to model elaborate memory safety properties. For example, separation logic [9] and its under-approximate variants [10] [11] provide intricate machinery [12] to deal with pointer reasoning and dynamic memory allocations. Here we demonstrate that under-approximate reasoning can also be applied to other families of properties as well without requiring the more elaborate separation semantics for programming languages which primary concern is not memory safety.

4 Relational Adversarial Logic

Adversarial Logic (AL) is a concurrent incorrectness logic used to under-approximate verification [13] [2] of security properties. To do so, it marries under-approximation of program behaviors with an adversarial model [14], where the attacker logic is encoded in the program logic itself, rather than being left as a meta-logic description. In particular, in this paper we focus on a model where the adversary is an arbiter of two or more program executions. As such, analysis of relational properties is encoded within the adversarial program, whose role is to execute external routines and compare results based on controlled assumptions.

It will be made clear in the following section that it is sufficient to derive the adversarial relational rules from the non-relational ones for ease of logic reasoning. As such, there is no need to create a new logic solely for the purpose of relational incorrectness reasoning in the adversarial setting. With careful encoding, the original adversarial logic is sufficient to under-approximate satisfaction of hyper-properties.

4.1 Target language

We will work with this toy imperative language made of variables, arrays, expressions, channels, predicates and commands. Expressions are made of named variables $x$, concrete integers $n$, symbolic values $\alpha$, random values, and their arithmetic combinations. Channel variables are named resources ($s_1$, $s_2$, etc) whose values are ordered lists of scalar values. In particular, the value $(s::x)$ is the concatenation of values in channel $s$ with the value of variable $x$ added to the end of $s$. The value $(s\ \setminus x)$ is the value of channel $s$ after removing the value of $x$ from the head. The value of an empty channel is an empty list.

Typical scalar variable types $T$ are available, which are sufficient to cover examples in this paper. Machine encoding of such data types is not central to the logic and remains out of scope for this paper. We will sometimes treat the $\text{rand}()$ value as an $\text{uint8}$ or a $\text{float}$ (as in the equivalence testing example of table 6). We will adopt $\text{rand8}()$ or $\text{randf}()$ as needed to make precise which version is used, or simply $\text{rand}()$ when this is obvious from context.
Variables  \( V ::= x \mid n \mid \alpha \)

Arrays  \( A ::= x[V] \)

Expressions  \( E ::= A \mid V \mid \text{rand()} \mid E + E \mid E - E \mid ... \)

Channels  \( L ::= s \mid \emptyset \mid (V::L) \mid (L::V) \mid (s \setminus V) \)

Predicates  \( B ::= B \land B \mid B \lor B \mid \neg B \mid E == E \mid E \leq E \mid ... \)

Data types  \( T ::= \text{uint8} \mid \text{uint32} \mid \text{float} \mid \text{string} \)

Commands  \( C ::= \text{skip} \mid x := E \mid x[v] := E \mid s := L \mid C_1 ; C_2 \mid C_1 || C_2 \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \mid \text{while } B \text{ do } C \text{ done} \mid \text{read}(s, x) \mid \text{write}(s, E) \mid \text{adv_assert}(B) \mid T x = E \text{ in } C \mid \text{Com}(C_1,C_2) \)

Predicates are built from the usual logical and, or, not, as well as equality and inequality tests. All commands can be used in program or adversarial terms, except assertions which are limited to the adversary. The communication primitive \( \text{Com} \) is distinct from read and write, and \( \text{Com} \) can be applied at any time after the corresponding write is completed and before the corresponding read is performed. This flexibility makes AL able to encode any desired input/output behavior. The study of specific IO caching strategies are out of scope for this paper, and we leave \( \text{Com} \) implicit in our examples as to apply the corresponding proof rule when required to make progress.

An addition to the object language used in the original (non-relational) AL presentation [1] is the ability to define strings, which are treated as initialized arrays \( \text{string } x = [x_1, ..., x_n] \) where \( n \) is a scalar integer value and \( [x_1, ..., x_n] \) is the array value. This is equivalent to nesting \( n \) scalar definitions in the base language.

\[
\text{string } x = [0, 1, 2, ..., n] \text{ in } c \overset{\text{def}}{=} \text{uint8 } x_0 = 0 \text{ in (uint8 } x_1 = 1 \text{ in ... (uint8 } x_n = n \text{ in c})}
\]

When copying between arrays, we adopt the following syntactic sugar to manipulate elements at specific array index as a set of indexed variables. A string or array of length \( n \) is represented as a list of \( n \) independent variables. A substring can be extracted from a bigger string by specifying the beginning or ending index of the desired sub-array.

\[
x[n] = v \overset{\text{def}}{=} x_n = v \\
x[l] \overset{\text{def}}{=} [x_0, x_1, ..., x_l] \\
x[l:] \overset{\text{def}}{=} [x_l, x_{l+1}, ..., x_n]
\]

This syntax is used to simplify our examples on URL parsing confusion vulnerability in the next section, which involves strings. For the sake of this article, it is assumed that arrays are accessed within bounds, as we wish to study properties other than memory safety. We refer to [10] [11] as the more suitable way to encode incorrectness for memory safety issues. Convenient syntactic sugar is also defined for conditionals:

\[
\text{if } P c \overset{\text{def}}{=} \text{if } P \text{ then } c \text{ else skip} \\
T x = E_1, y = E_2 \text{ in } C \overset{\text{def}}{=} T x = E_1 \text{ in } (T y = E_2 \text{ in } C) \\
\text{if } P c_1 \text{ else if } Q c_2 \text{ else } c_3 \overset{\text{def}}{=} \text{if } P \text{ then } c_1 \text{ else if } (Q \text{ then } c_2 \text{ else } c_3)
\]
Reading or writing string values on a channel are equivalent to reading or writing one element at a time, therefore it is sufficient to give axioms for reading and writing singletons, which can express longer sequences of input/output.

\[
\text{len} = \text{read}(s, \text{str}) \overset{\text{def}}{=} \text{read}(s, \text{str}[0]); \ldots ; \text{read}(s, \text{str}[n-1]); \text{len} = n;
\]

\[
\text{len} = \text{write}(s, \text{str}) \overset{\text{def}}{=} \text{write}(s, \text{str}[0]); \ldots ; \text{write}(s, \text{str}[n-1]); \text{len} = n;
\]

AL can leverage the full adversarial term when available, however it is not required and a minimal template is often sufficient. Example 2 and 3 of section 5 show how such templates are the basis for attack proofs.

### 4.2 Adversarial reasoning

AL inherits from incorrectness logic in that its semantics is defined using a couple of relations \(\text{ok}\) and \(\text{ad}\) where \(\text{ok}\) is the program interpretation and \(\text{ad}\) is the adversarial interpretation. We recall that inference triples in incorrectness logic are written as:

\[
\begin{align*}
\text{ok} : P & \quad \text{c} & \quad \text{er} : R & \overset{\text{def}}{=} [\text{ok} : P][\text{er} : R] \\
\text{ok} : P & \quad \text{c} & \quad \text{ok} : Q & \overset{\text{def}}{=} [\text{ok} : P]\text{c}[\text{ok} : Q].
\end{align*}
\]

Each code fragment \(\text{c}\) lifts precondition \(P\) to postcondition \(Q\) and error postcondition \(R\). We generalize \(\text{er}\) by \(\text{ad}\) so that inference triples are written as:

\[
\begin{align*}
\text{ok} : P_1 & \quad [\text{ad} : P_2] \quad \text{c} & \quad \text{ok} : Q_1 & \quad [\text{ad} : Q_2] & \overset{\text{def}}{=} [\text{ok} : P_1]\text{c}[\text{ok} : Q_1] & \text{and} [\text{ad} : P_2]\text{c}[\text{ad} : Q_2].
\end{align*}
\]

Program interpretation and adversarial interpretation are compositional and allow independent reasoning over \(\text{ok}\) and \(\text{ad}\). A novelty of adversarial logic is that assertions are solely checked by the adversary. Rules Success and Failure check the satisfiability of an attack contract rather than a program contract. Checking of assertions augment adversarial knowledge, as both outcomes may inform the choice of subsequent interactions with the program.

\[
\begin{align*}
\text{Success} & - \quad [\text{ad} : Q \land (Q \Rightarrow B)] \quad \text{adv_assert}(B) & \quad [\text{ad} : Q \land \text{true}] \\
\text{Failure} & - \quad [\text{ad} : Q \land (Q \Rightarrow \neg B)] \quad \text{adv_assert}(B) & \quad [\text{ad} : Q \land \neg B]
\end{align*}
\]

More succinctly, rules in AL are written as:

\[
[\epsilon : P] \quad \text{c} \quad [\epsilon : Q]
\]

where \(\epsilon \in \{\text{ok}, \text{ad}\}\) is a short notation to write two rules \([\text{ok} : P] \quad \text{c} \quad [\text{ok} : Q]\) and \([\text{ad} : P] \quad \text{c} \quad [\text{ad} : Q]\) as one when the rule is valid in both program and adversarial interpretations. This allows a much more succinct representation of adversarial logic proof rules.

Basic operations such as reading and writing on channels require the use of new Read and Write rules. I/O rules are defined as synchronous primitives, where a read (resp. write) happens immediately if data is available on the (potentially infinite) channel.
Access to channel are implemented using Floyd’s axiom of assignment as applied to channel values (lists). Channels in AL are accessed first-in / first-out and can be used to represent files, sockets, and other inter-process communication primitive of real systems. If an attempt is made to read data on an empty channel, the Skip rule can be used to simulate a blocking read. Reads and writes are performed one datum at a time. Operations on bigger data length can easily be encoded using repetition of these base rules.

Parallel composition of a program $p$ and an adversary $a$ is constructed from a program interpretation and an adversarial interpretation of parallel terms:

$$\text{Par} \quad \epsilon_1 : P_1 |(\epsilon_2 : P_2) \quad \epsilon_1, \epsilon_2 \in \{\text{ok}, \text{ad}\} \quad \epsilon_1 : \exists s. P(s) \land s = (s') \land x = v$$

The Par rule does not permit communication in itself. This follows from the adversarial logic principle that no information is shared unless explicitly revealed. This parallel rule is unusual as it uses two pre-conditions and two post-conditions, enforcing variable separation without requiring an extra conjunctive connector as done in separation logic [9].

When two parallel terms need to share information, AL requires to use the communication rule Com on channel $s$. While program and adversary share no local or free variables, shared channels are required for communication. To preserve uniqueness of names, we may use $s_a$ in the adversarial interpretation, and $s_p$ in the program interpretation to refer to channel $s$, although we will just use $s$ when the meaning is clear from context. Examples of Com usage can be found in the simplest example of next section in table 3.

$$\text{Com} \quad \epsilon_1 : P(\epsilon_2 : A) \quad \epsilon_1, \epsilon_2 \in \{\text{ok}, \text{ad}\} \quad \epsilon_1 : \exists s. P(s) \land s = (s') \land x = v$$

Applications of adversarial logic include cases where the adversary wishes to infer hidden values of variables and predicates that have not been communicated. In these cases, the Adversarial Consequence rule can augment the adversarial postcondition $A'$ if observable values communicated by the program are consequences of hidden program conditions, represented as program predicate $Q$. We require that Free($Q$) = $\emptyset$ as free variables in $Q$ are not defined in the adversarial term. This can be guaranteed by creating fresh names during the introduction of $Q$ in the adversarial context, so that no names are shared. It is assumed that $s \in Chan(A) \cap Chan(P)$. Basic usage of this rule can be found in all examples of section 5.
(a) Adversarial logic core rules with restored symmetry from $\text{IL} : \epsilon \in \{\text{ok, ad}\}$

$$
\begin{align*}
\text{Unit} & \quad [e : P] \text{skip} [e : P'] & \text{Constancy} & \quad [e : P][e : Q] & \text{Mod}(c) \cap \text{Free}(F) = \emptyset \\
\text{Consequence} & \quad [e : P \Rightarrow P'] & [e : P][e : Q] & \quad [e : Q' \Rightarrow Q] \\
\text{Assume} & \quad [e : P] \text{assume}(B) & [e : P \land B] & \quad [e : P] x = \text{rand}(c) [e : 3x'.P(x'/x) \land x = v] \\
\text{Rand} & \quad [e : P] x = e & [e : 3x'.P(x'/x) \land x = e(x'/x)] \\
\text{Assign} & \quad [e : P_1][e : Q_1] & [e : P_2][e : Q_2] & \quad [e : P \land x = e][e : Q] x \notin \text{Free}(P) \\
\text{Disj} & \quad [e : P][e : Q] & [e : Q][e : R] & \quad \text{Local} \quad [e : P][e : Q] x = e \in c \quad [e : 3x \in T.Q] \\
\text{Seq} & \quad [e : P][e : Q][e : R] & \quad \text{Choice} \quad [e : P][e : Q] \epsilon_1, \epsilon_2 \in \{\text{ok, ad}\} \\
\text{Choice} & \quad \text{Iterate Zero} \quad [e : P]^\ast & \quad \text{Iterate non-zero} \quad [e : P]^* \epsilon_1[e : Q] \\
\end{align*}
$$

while $B \text{ do C done } \subseteq (\text{assume}(B); C)^\ast; \text{assume}(\neg B)$

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

(b) Adversarial Logic: communication rules between program and adversary

$$
\begin{align*}
\text{Read} & \quad [e : P] \text{read}(s, x) [e : \exists v \exists x'.\exists s'.P(s'/s, x'/x) \land s = (s'\backslash v) \land x = v] \\
\text{Write} & \quad [e : P] \text{write}(s, x) [e : \exists v \exists s'.P(s'/s) \land s = (s'\backslash v)] \\
\text{Par} & \quad [e_1 : P_1][e_2 : Q_1] & [e_2 : P_2][e_2 : Q_2] & e_1, e_2 \in \{\text{ok, ad}\} \\
\text{Com} & \quad [e_1 : P][e_2 : A] & e_2 \in \{\text{ok, ad}\} & e_1 \in \{\text{ok, ad}\} \\
\end{align*}
$$

(c) Adversarial Logic: knowledge rules between program and adversary

$$
\begin{align*}
\text{PBV} & \quad \text{ok}(P(n))[\text{ad A}(m)] c_{\text{p}} [\text{ok}: P(n + i)][\text{ad A}(m + j)] c_{\text{a}} [\text{ok}: \exists n.P(n)][\text{ad A}(m)], i, j \in \{0, 1\} \land i + j \geq 1 \\
\text{Adv.Cons.} & \quad \text{ok}(P')[\text{ad A}] c_{\text{p}} [\text{ok}: \exists v_1.Q \land \exists v_2.Q \land \exists v_1 = v_2] \\
\text{Success} & \quad [\text{ad Q} \land (Q \Rightarrow B)] \text{adv assert}(B) [\text{ad Q} \land \text{true}] \\
\text{Failure} & \quad [\text{ad Q} \land (Q \Rightarrow \neg B)] \text{adv assert}(B) [\text{ad Q} \land \neg B]
\end{align*}
$$
Adversarial logic generalizes the Backward variant rule of incorrectness logic [2] for parallel composition of program and adversarial terms, which we name the Parallel Backward Variant, or PBV.

\[
\text{PBV} \quad [ok : P(n)][\text{ad : A}(m)] c_p \parallel c_a [ok : P(n + i)][\text{ad : A}(m + j)] \\
[ok : P(0)][\text{ad : A}(0)] c_p^0 \parallel c_a^0 [ok : \exists n. P(n)][\text{ad : \exists m. A}(m)]
\]

AL’s backward variant rule is a parallel composition of a program fragment \(c_p\) repeated \(n\) times with an adversarial code fragment \(c_a\) repeated \(m\) times. It is not required for the number of program steps \(n\) and adversarial steps \(m\) to be the same, as long as at least one step is taken at each iteration \((i, j) \in \{0, 1\} \land i + j \geq 1\). This condition enforces that every extracted attack trace is finite.

The PBV rule cannot be expressed using two instances of IL’s original BV rule, as PBV can express conditions where adversarial and program conditions are subject to communication. It may also be useful to apply the original sequential BV rule in parallel when program and adversarial terms are independently reducible, however this does not equate to using the parallel version of the rule which can provide synchronization across terms. A practical example of PBV usage is demonstrated in table 4 of section 5.

One notable incorrectness rule absent from AL is the sequential short-circuit. In this rule, execution of the second term of a sequence is avoided if the first term terminates by an error. As adversarial logic is meant to analyze consequences of erroneous program executions, short-circuiting serves no benefit. This highlights a key difference between IL’s original error relation \(err\) and AL’s adversarial relation \(ad\). All other rules of adversarial logic are similar to incorrectness logic with the difference that either \(ok\) or \(ad\) can be used in the precondition, therefore restoring a lost symmetry in incorrectness logic while preserving its meaning.

### 4.3 Relational rules

We now introduce the adversarial relational rules. These new relational rules are derived: they are constructed from simpler existing logic rules of adversarial logic, rather than being introduced as ground axioms.

Fig. 2: Relational Assertion rule RelFalse where PSF = Par + Seq \(\times\) 2 + Failure
Relational Adversarial Logic

Relational assertions are the basis for checking hyper-properties comparing program executions $c_1$ and $c_2$. The role of the adversarial term is then to collect results of similarities or differences of outcome for $c_1$ and $c_2$. Exhibiting such relational assertions in the original formulation of AL allows us to inherit the attack soundness theorem for free.

Note how the definition of term $a_{s,r}$ is key: the result of $c_1$ ends up in $r_1$ while the result of $c_2$ ends up in $r_2$, which are then compared in the adversarial assertion. However, note that the same $\alpha$ symbolic value is used in both cases: this makes sure that we exhibit a divergence of result between $c_1$ and $c_2$ on the same input. We will see in section 5 how such characterization is useful to extract parser differentials such that the one encountered in URL parsing libraries.

5 Under-approximate Relational Analysis In AL

In this section, we introduce three case studies for relational AL. The first example in subsection 5.1 is a simple relational analysis between two capture the flag programs, and is intended as a first exposure to the usage of AL. The second example in subsection 5.2 exhibits the full proof of our motivational example of URL parsing confusion from section 3. This second example is an instance of disagreement property between two implementations. Finally, our third example in subsection 5.3 demonstrates a use-case for which under-approximate agreement is desired, where two instances of a market pricing functions are compared for common points.

5.1 Example 1: Simple Relational Case

Let us consider the example in table 3 where an adversary wants to toggle the flag $win$ to 1 with an input value $n$ reaching value 10 million (10M for short) in the first program, and value 20 million (20M for short) in the second program. We demonstrate the existence of a value for $\alpha$ for which both programs differ.

An adversarial proof such as the one in table 3 may contain several proof phases corresponding to non-blocking subsequences of program or adversarial derivations. A proof is typically divided into the following phases:

1. The bootstrap phase ($P_0$ to $P_1$, $Q_0$ to $Q_1$, and $A_0$ to $A_2$) where programs under analysis and adversarial program have yet to be composed.
Table 3: Simple example of hyper-property checking and its relational adversarial proof.

<table>
<thead>
<tr>
<th>Pre: $s_1$ est.</th>
<th>Pre: $s_2$ est.</th>
<th>Precond: $s_1$ and $s_2$ est.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. prog1(int $s_1$)</td>
<td>11. prog2(int $s_2$)</td>
<td>21. adversary(int $s_1$, int $s_2$)</td>
</tr>
<tr>
<td>2. {</td>
<td>12. {</td>
<td>22. {</td>
</tr>
<tr>
<td>3. uint32 $n$, win in</td>
<td>13. uint32 $n$, win in</td>
<td>23. uint32 res1 = 0 in</td>
</tr>
<tr>
<td>4. read($s_1$, n);</td>
<td>14. read($s_2$, n);</td>
<td>24. uint32 res2 = 0 in</td>
</tr>
<tr>
<td>5. if ($n &gt; 10M$)</td>
<td>15. if ($n &gt; 20M$)</td>
<td>25. write($s_1$, $\alpha$);</td>
</tr>
<tr>
<td>6. win = 1;</td>
<td>16. win = 1;</td>
<td>26. write($s_1$, res1);</td>
</tr>
<tr>
<td>7. else</td>
<td>17. else</td>
<td>27. write($s_2$, $\alpha$);</td>
</tr>
<tr>
<td>8. win = 0;</td>
<td>18. win = 0;</td>
<td>28. write($s_2$, res2);</td>
</tr>
<tr>
<td>9. write($s_1$, win);</td>
<td>19. write($s_2$, win);</td>
<td>29. adv_assert(res1 $\neq$ res2);</td>
</tr>
<tr>
<td>10. }</td>
<td>20. }</td>
<td>30. }</td>
</tr>
</tbody>
</table>

→ Local
Adversary(int $s_1$, int $s_2$) { |
→ Local $\times$ 2
uint32 res1 = 0, res2 = 0 |
→ Write
write($s_1$, $\alpha$) |
→ Com
$A_2 = \{\exists s_2. A_2(s_2/s_1) \land s_1 = (\alpha)\}$ |
→ Read
read($s_1$, res1) |
$A_0 = \{\exists n, A_0(n = m) \land (n > 10M \land win = 1) \lor (n \leq 10M \land win = 0)\} \land win = res1$ |
→ Write
write($s_2$, $\alpha$) |
→ Com
$A_3 = \{\exists s_2, A_3(s_2/s_1) \land s_1 = (s_2^1/\alpha)\}$ |
→ Read
read($s_2$, res2) |
$A_0 = \{\exists n, A_0(n = m) \land ((n > 10M \land win = 1) \lor (n \leq 10M \land win = 0)) \land win = res1\}$ |
$A_1 = \{\exists n. A_1(n = m) \land ((n > 10M \land win = 1) \lor (n \leq 10M \land win = 0)) \land win = res1\}$ |
→ Conseq
$A_{12} = \{\exists m. P_{12}(m = 1) \land ret1 = 1 \land ret2 = 0\}$ |
→ Assert
assert(res1 $\neq$ res2) |
$A_{13} = \{\exists n. A_{13}(n = m) \land ((n > 10M \land win = 1) \lor (n \leq 10M \land win = 0)) \land win = res1\}$ |
→ Local
Prog1(int $s_1$) { |
→ Assign
uint32 m = 0, win = 0 |
→ Com
read($s_1$, m) |
→ Read
read($s_1$, m) |
→ I/Write
if ($m > 10M$) win = 1 |
→ Else/Write
else win = 0 |
→ Disj
P_8 = (ok $P_8 \lor P_9$) |
→ Write
write($s_1$, win) |
→ Com
$P_2 = \{ok \exists s_2, P_2(s_2/s_1) \land s_1 = (s_2/\alpha)\}$ |
→ Local
Prog2(int $s_2$) { |
→ Assign
uint32 m = 0, win = 0 |
→ Com
read($s_2$, m) |
→ Read
read($s_2$, m) |
→ I/Write
if ($m > 20M$) win = 1 |
→ Else/Write
else win = 0 |
→ Disj
$Q_5 = \{ok Q_5 \lor Q_6\}$ |
→ Write
write($s_2$, win) |
→ Com
$Q_2 = \{ok \exists s_2, Q_2(s_2/s_1) \land s_1 = (s_2/\alpha)\}$ |
2. The initial phase \((P_2, A_3)\) to \((P_7, A_6)\) typically starts with an application of the \textbf{Par} and \textbf{Com} rules to exchange messages until composed terms fail to make any progress other than \textbf{Skip}.

3. Subsequent intermediate phases separated by applications of the \textbf{Par} and \textbf{Com} rules for each program within the relational analysis (such as \(A_7\) to \(A_{11}\) and \(Q_2\) to \(Q_7\) for the second program under analysis).

4. The final phase ends by the attacker applying the \textbf{Conseq} and \textbf{Assert} rules where the adversarial assertion is checked for satisfiability \((A_{12}\) to \(A_{13}\)).

Note how symbolic variable \(\alpha\) is introduced by the adversarial interpretation \((A_2)\) and propagated to the target program’s logic \((P_2\) and \(Q_2\) respectively) using the communication rule. Program and adversarial interpretations remain independent until the parallel rule is used to compose terms.

It is important to understand that \(A_5\) and \(A_{10}\) are insufficient to prove the adversarial assertion \((\text{res1} \neq \text{res2})\) as conditions on \(\alpha\) are unknown until the \textbf{AdvCons} rule (adversarial consequence) is applied. This rule is typically applied once per \textbf{read} operation.

5.2 Example 2: URL parsing confusion vulnerability

Our second example demonstrates a proof of disagreement between two URL parsing implementations used in our motivational example of section 3. The attack proof is this example makes uses of the parallel backward variant (PBV) rule in table 4. A combination of PBV and disjunction rules allows the adversary to exhibit a target program iteration without requiring to unroll the loop \(O(n)\) times (with \(n\) the number of characters in the input string). The PBV rule is used twice, once per program under relational analysis \(P\) and \(Q\).

5.3 Example 3: Equivalence Testing For Pricing Functions

Our third example of relational analysis illustrates an \textit{agreement} property between several comparable function implementations. In the under-approximate setting, agreement is only checked for certain inputs rather than all possible inputs. In this case, we compare two implementations of a pricing service to find inputs for which pricing functions agree. Such boundary points are often used as an approximate financial yield analysis.

Consider the code in table 5 where a pricing service contains two functions \textit{GetPrice} and \textit{GetPrice2} reading on channels \(s_1\) and \(s_2\) to compute market price based on a globally initialized random market value \textit{initp} and ordered quantities \textit{num}. The first function converges faster than the other due to a different current price calculation.
Table 4: Relational attack proof for URL confusion vulnerability from table 2.

<table>
<thead>
<tr>
<th>Action</th>
<th>Proof</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local x 2</td>
<td>Adversary(int x1, int x2) {</td>
<td>Adversary(int x1, int x2) {</td>
</tr>
<tr>
<td>Write</td>
<td>A0 ≡ (ok, x1 = 0 ∧ x2 = 0)</td>
<td>A0 ≡ (ok, x1 = 0 ∧ x2 = 0)</td>
</tr>
<tr>
<td>Write</td>
<td>(q1, q2 in)</td>
<td>(q1, q2 in)</td>
</tr>
<tr>
<td>Par + Com</td>
<td>A2 = (ok, 3 2 ÷ A1[s1/q1] ∧ x1 = [n1, ... , n6])</td>
<td>A2 = (ok, 3 2 ÷ A1[s1/q1] ∧ x1 = [n1, ... , n6])</td>
</tr>
<tr>
<td>Assign</td>
<td>A3 ≡ (ok, 3 2 ÷ A2[s2/q2] ∧ x2 = 0)</td>
<td>A3 ≡ (ok, 3 2 ÷ A2[s2/q2] ∧ x2 = 0)</td>
</tr>
<tr>
<td>Read</td>
<td>A4 ≡ (ok, 3 2 ÷ A3[s3/q3] ∧ x1 = [h1])</td>
<td>A4 ≡ (ok, 3 2 ÷ A3[s3/q3] ∧ x1 = [h1])</td>
</tr>
<tr>
<td>Write</td>
<td>(s4/q1, x1)</td>
<td>(s4/q1, x1)</td>
</tr>
<tr>
<td>Par + Com</td>
<td>A5 = (ok, 3 2 ÷ A4[s4/q4] ∧ x1 = 0 ∧ q1 = h1)</td>
<td>A5 = (ok, 3 2 ÷ A4[s4/q4] ∧ x1 = 0 ∧ q1 = h1)</td>
</tr>
<tr>
<td>Assign</td>
<td>A6 ≡ (ok, 3 2 ÷ A5[s5/q5] ∧ x1 = s4 ÷ s1)</td>
<td>A6 ≡ (ok, 3 2 ÷ A5[s5/q5] ∧ x1 = s4 ÷ s1)</td>
</tr>
<tr>
<td>Write</td>
<td>(s6/q2, x2)</td>
<td>(s6/q2, x2)</td>
</tr>
<tr>
<td>Par + Com</td>
<td>A7 = (ok, 3 2 ÷ A6[s6/q6] ∧ x2 = [n1, ... , n6])</td>
<td>A7 = (ok, 3 2 ÷ A6[s6/q6] ∧ x2 = [n1, ... , n6])</td>
</tr>
<tr>
<td>Com</td>
<td>A8 ≡ (ok, 3 2 ÷ A7[s7/q7] ∧ x2 = [h2])</td>
<td>A8 ≡ (ok, 3 2 ÷ A7[s7/q7] ∧ x2 = [h2])</td>
</tr>
<tr>
<td>Read</td>
<td>(s8/q3, x2)</td>
<td>(s8/q3, x2)</td>
</tr>
<tr>
<td>Adv.Conseq.</td>
<td>A10 = (ok, A9 ∧ url♯ = [n1, ... , n6] = url♯ + 3i, 2 &lt; i &lt; 2 ÷ url♯)</td>
<td>A10 = (ok, A9 ∧ url♯ = [n1, ... , n6] = url♯ + 3i, 2 &lt; i &lt; 2 ÷ url♯)</td>
</tr>
<tr>
<td>❍ Success A1 = (ok, A10 &amp; truly)</td>
<td>❍ Success A1 = (ok, A10 &amp; truly)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action</th>
<th>Proof</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>parseURL(int x2) {</td>
<td>parseURL(int x2) {</td>
</tr>
<tr>
<td>Write</td>
<td>P0 ≡ (ok, x2 = 0)</td>
<td>P0 ≡ (ok, x2 = 0)</td>
</tr>
<tr>
<td>Local x 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>P1 ≡ (ok, x1 = 0 ∧ URL = 8 ∧ host = 8 ∧ i = 0 ∧ p1 = 0 ∧ delim = 0 ∧ len = 0)</td>
<td>P1 ≡ (ok, x1 = 0 ∧ URL = 8 ∧ host = 8 ∧ i = 0 ∧ p1 = 0 ∧ delim = 0 ∧ len = 0)</td>
</tr>
<tr>
<td>Par + Com</td>
<td>P2 ≡ (ok, 3 2 ÷ P1[s1/q1] ∧ x1 = [n1, ... , n6])</td>
<td>P2 ≡ (ok, 3 2 ÷ P1[s1/q1] ∧ x1 = [n1, ... , n6])</td>
</tr>
<tr>
<td>Read x n + Assign</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>(s4/q1, x1)</td>
<td>(s4/q1, x1)</td>
</tr>
<tr>
<td>Par + Com</td>
<td>A5 = (ok, 3 2 ÷ A4[s4/q4] ∧ x1 = 0 ∧ q1 = h1)</td>
<td>A5 = (ok, 3 2 ÷ A4[s4/q4] ∧ x1 = 0 ∧ q1 = h1)</td>
</tr>
<tr>
<td>Assign</td>
<td>A6 ≡ (ok, 3 2 ÷ A5[s5/q5] ∧ x1 = s4 ÷ s1)</td>
<td>A6 ≡ (ok, 3 2 ÷ A5[s5/q5] ∧ x1 = s4 ÷ s1)</td>
</tr>
<tr>
<td>Write</td>
<td>(s6/q2, x2)</td>
<td>(s6/q2, x2)</td>
</tr>
<tr>
<td>Par + Com</td>
<td>A7 = (ok, 3 2 ÷ A6[s6/q6] ∧ x2 = [n1, ... , n6])</td>
<td>A7 = (ok, 3 2 ÷ A6[s6/q6] ∧ x2 = [n1, ... , n6])</td>
</tr>
<tr>
<td>Com</td>
<td>A8 ≡ (ok, 3 2 ÷ A7[s7/q7] ∧ x2 = [h2])</td>
<td>A8 ≡ (ok, 3 2 ÷ A7[s7/q7] ∧ x2 = [h2])</td>
</tr>
<tr>
<td>❍ Success A1 = (ok, A8 &amp; truly)</td>
<td>❍ Success A1 = (ok, A8 &amp; truly)</td>
<td></td>
</tr>
</tbody>
</table>
Table 5: Two pricing functions with user-supplied order number and random initial price.

```c
// Preprocessor definitions
1. define V1MIL 1000000
2. define V9MIL (V1MIL*9)
3. define V18MIL (V1MIL*18)
4. define V10MIL (V1MIL*10)
5. define V20MIL (V1MIL*20)

// Shared initial service state
6. float initp = rand();

// Precond: chan s1 established
7. void GetPrice(int s1)
8. {
9. float curp in
10. uint32 ord in
11. float dec in
12. while (true) do
13. read(s1, ord);
14. dec = ord / V10MIL;
15. if (ord <= V9MIL)
16. curp = initp * (1 - dec);
17. else
18. curp = initp / 10;
19. write(s1, curp);
20. done
21. }

// Precond: chan s1 and s2 established
22. void GetPrice2(int s2)
23. {
24. float curp2 in
25. uint32 ord2 in
26. float dec2 in
27. while (true) do
28. read(s2, ord2);
29. dec2 = ord2 / V20MIL;
30. if (ord2 <= V18MIL)
31. curp2 = initp * (1 - dec2);
32. else
33. curp2 = initp / 10;
34. write(s2, curp2);
35. done
36. }
```

In order to model this example in adversarial logic, we define and use a derived rule $\text{Dup}$ at step $(P_2, A_2)$ in table 6. The $\text{Dup}$ rule can be expressed solely based on the parallel rule with parameters $\epsilon_1 = \epsilon_2 = ok$ and $P_1 = P_2$ and $Q_1 = Q_2$ and $c_1 = c_2$.

\[
\text{Dup} \quad \left[ \begin{array}{c}
ok : P_1 \\
ok : Q_1 \\
ok : P_1 \\
ok : Q_1 \\
ok : P_1 \\
ok : P_1 \\
ok : Q_1 \\
ok : Q_1
\end{array} \right] \quad \left[ \begin{array}{c}
ok : P_1 \\
ok : Q_1 \\
ok : P_1 \\
ok : Q_1 \\
ok : P_1 \\
ok : P_1 \\
ok : Q_1 \\
ok : Q_1
\end{array} \right] 
\]

\[
\left[ \begin{array}{c}
ok : P_1 \\
ok : P_1 \\
ok : Q_1 \\
ok : Q_1
\end{array} \right] 
\]
Table 6: Equivalence testing: PBV and adversarial consequence rules are combined to find an input for which two pricing functions have the same output.

| P0 = (ok #) | A0 = (ok 0) | float guess1; | A1 = (ok v1 guess1 = v1 ∧ A0) | float guess2; | A2 = (ok v2 guess2 = v2 ∧ A1) | → Loc |
| P1 = (ok $f instp = rand); | → Rand float instp = rand(); | P2 = (ok $f instp = f ∧ P1); | → Loc |

| float guess2; | → Loc |

| P3 = (ok A2 guess2 = v2 ∧ A1); | → Loc |

| P4 = (ok A1 guess1 = v1 ∧ A0); | → Loc |

| P5 = (ok A0 guess1 = v1 ∧ A0); | → Loc |

| P6 = (ok A0 guess2 = v2 ∧ A1); | → Loc |

| P7 = (ok A1 guess1 = v1 ∧ A0); | → Loc |

| P8 = (ok A0 guess2 = v2 ∧ A1); | → Loc |

| P9 = (ok A2 guess2 = v2 ∧ A1); | → Loc |

| P10 = (ok A1 guess1 = v1 ∧ A0); | → Loc |

| P11 = (ok A0 guess2 = v2 ∧ A1); | → Loc |

| P12 = (ok A2 guess2 = v2 ∧ A1); | → Loc |

| P13 = (ok A1 guess1 = v1 ∧ A0); | → Loc |

| P14 = (ok A0 guess2 = v2 ∧ A1); | → Loc |

| P15 = (ok A2 guess2 = v2 ∧ A1); | → Loc |

| P16 = (ok A1 guess1 = v1 ∧ A0); | → Loc |

| P17 = (ok A0 guess2 = v2 ∧ A1); | → Loc |

| P18 = (ok A2 guess2 = v2 ∧ A1); | → Loc |

| P19 = (ok A1 guess1 = v1 ∧ A0); | → Loc |

| P20 = (ok A0 guess2 = v2 ∧ A1); | → Loc |

| P21 = (ok A2 guess2 = v2 ∧ A1); | → Loc |

| P22 = (ok A1 guess1 = v1 ∧ A0); | → Loc |

| P23 = (ok A0 guess2 = v2 ∧ A1); | → Loc |

| P24 = (ok A2 guess2 = v2 ∧ A1); | → Loc |

| P25 = (ok A1 guess1 = v1 ∧ A0); | → Loc |

| P26 = (ok A0 guess2 = v2 ∧ A1); | → Loc |

| P27 = (ok A2 guess2 = v2 ∧ A1); | → Loc |

| P28 = (ok A1 guess1 = v1 ∧ A0); | → Loc |

| P29 = (ok A0 guess2 = v2 ∧ A1); | → Loc |

| P30 = (ok A2 guess2 = v2 ∧ A1); | → Loc |

| P31 = (ok A1 guess1 = v1 ∧ A0); | → Loc |

| P32 = (ok A0 guess2 = v2 ∧ A1); | → Loc |

| P33 = (ok A2 guess2 = v2 ∧ A1); | → Loc |
Similarly to our previous example, we use the parallel backwards variant rule once per analyzed program to speed up the analysis. Unlike the previous example however, we look for points of agreement rather than points of disagreement between functions, so that we can discover boundary values of specific pricing function intervals. The details of our adversarial proof is given in table 6.

Extending this logic, one could extract every single boundary values by making sure subsequent values for $\alpha$ are excluded from the search, in a similar way one would extract every possible model of a satisfiable formula using a constraint solver. For this, one additional loop would be necessary around the adversarial program to enumerate all such boundary values. This enumeration is not unrolled for conciseness, still looks promising as a future direction to enable under-approximate functional analysis via adversarial reasoning.

6 Semantics

We now present a denotational semantics for adversarial logic in table 7. This semantics is defined compositionally for each of the rules of AL.

<table>
<thead>
<tr>
<th>Table 7: Relational denotational semantics for AL with transitions from state pairs $(\sigma_p, \sigma_a)$ to $(\sigma_p, \sigma_a)$ with $\epsilon, \epsilon_1, \epsilon_2 \in {ok, ad}$ and $\Sigma_x \in {\Sigma_a, \Sigma_p}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma_a: [\text{Variables} \rightarrow \text{Values}]$</td>
</tr>
<tr>
<td>$\Sigma_p: [\text{Variables} \rightarrow \text{Values}]$</td>
</tr>
<tr>
<td>$H = \Sigma_p \times \Sigma_a$</td>
</tr>
<tr>
<td>$[B] \in \Sigma_p \rightarrow \text{Bool}$</td>
</tr>
<tr>
<td>$\sigma = (\sigma_p, \sigma_a) \in H$</td>
</tr>
<tr>
<td>$[C] \subset H \times H$</td>
</tr>
</tbody>
</table>

Note how the state pair $(\sigma_p, \sigma_a)$ is employed to represent the parallel composition of the program and adversarial states. The fact that programs and adversary share no variables render the semantics of parallel composition a simple disjoint union of the semantics of individual programs. As several programs are
composed with the adversarial term for relational analysis, we simply represent
\( \sigma_p \) as \( \sigma_{p1} \times \ldots \times \sigma_{pn} \) where variables appearing in each program state are only
defined for this part of the composition.

We now prove soundness of the logic and semantics by first reminding some
standard definitions.

**Definition 1 (Post and semantic triples).** For any relation \( r \in \Sigma \times \Sigma \) and
predicate \( p \subseteq \Sigma \):

- The post-image of \( r \), \( \text{post}(r) \in \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma) : \text{post}(r)p = \{ (\sigma' | \exists \sigma \in p, (\sigma, \sigma') \in r \} 
- The over-approximate Hoare triple: \( \{ p \} r \{ q \} \) iff \( \text{post}(r)p \subseteq q \)
- The under-approximate incorrectness triple: \( \{ p \} r[q] \) is true iff \( \text{post}(r)p \supseteq q \)

We then introduce adversarial semantic triples, which can be understood as
a composition of semantic relation between program states \( \Sigma_p \) and adversarial
states \( \Sigma_a \) where \( \Pi = \Sigma_p \times \Sigma_a \) is the decomposed view of the state space.

**Definition 2 (Adversarial triples).** For any composed relation \( \langle \text{ok}, \text{ad} \rangle \in \Pi \times \Pi \) and predicates \( (p, a) \subseteq \Pi \) with \( p \) the program predicate and \( a \) the adver-
sarial predicate:

- The post-image of \( \langle \text{ok}, \text{ad} \rangle \) noted \( \text{post}(\langle \text{ok}, \text{ad} \rangle) \in \mathcal{P}(\Pi) \rightarrow \mathcal{P}(\Pi) : 
\text{post}(\langle \text{ok}, \text{ad} \rangle)(p, a) = \{ (\sigma_q, \sigma_b) | \exists (\sigma_p, \sigma_a) \in (p, a). (\sigma_p, \sigma_a), (\sigma_q, \sigma_b) \in \langle \text{ok}, \text{ad} \rangle \} 
- The under-approximate adversarial triple:
\[ \{ p \}[a] \langle \text{ok}, \text{ad} \rangle[q][b] \text{ is true iff } \text{post}(\langle \text{ok}, \text{ad} \rangle)(p, a) \supseteq (q, b) \]

Conditions for membership \( ((\sigma_p, \sigma_a), (\sigma_q, \sigma_b)) \in \langle \text{ok}, \text{ad} \rangle \) are defined as:

1. \( (\sigma_p, \sigma_q) \in \text{ok} \) and \( (\sigma_a, \sigma_b) \in \text{ad} \) if \( \text{VAR}(\sigma_p) \cap \text{VAR}(\sigma_a) = \emptyset \)
2. \( \langle \text{ok}, \text{ad} \rangle((\sigma_p, \sigma_a)) = (\sigma_q, \sigma_b) \) otherwise.

The first formulation of membership is enough for the **Par** rule and all rules
where program and adversary are reduced independently. The second formulation
is needed for the **Com**, **Backward variant** and **Adversarial consequence** rules as a channel \( s \) may ve involved to share information between
program and adversary.

**Definition 3 (Incorrectness principles in adversarial logic).** Adversarial
logic preserves the symmetries of incorrectness logic:

- \( \land \lor \) symmetry: \( [e: p]c[e: q_1] \land [e: p]c[e: q_2] \iff [e: p]c[e: q_1 \lor q_2] \)
- \( \uparrow \downarrow \) symmetry: \( [e: p \Rightarrow p'] \land [e: p]c[e: q] \land [e: q' \Rightarrow q] \iff [e: p']c[e: q'] \)
Adversarial logic inherits the consequence and disjunction rules of incorrectness logic, and therefore preserves incorrectness symmetries. Under-approximate reasoning is similarly unchanged, preserving principles of agreement and denial [2]. The central tool for soundness proof is the characterization lemma, which relates the state transition system of the denotational semantics to the inference system of adversarial logic.

**Lemma 1 (Characterization).** The following statements are equivalent:

1. \([\text{pre}_{\text{ok}}: p]\text{[pre}_{\text{ad}}: a] C_1 || C_2 \text{[post}_{\text{ok}}: q]\text{[post}_{\text{ad}}: b]\) is true.

2. Every state in the conclusion is reachable from a state in the premises:
\[\forall (p, q) \exists p \in p \exists a \in a : ((\sigma_p, \sigma_a), (\sigma_q, \sigma_b)) \in (\text{ok}, \text{ad})\]

The characterization lemma in Adversarial Logic extends the one of Reverse Hoare Logic of de Vries and Koutavas [13] as inherited by Incorrectness Logic [2]. Sufficient conditions for the characterization lemma to hold can be decomposed into three subcases:

1. if \(\sigma_a = \sigma_b\):
\[\forall (p, q) \exists p \in p : (\sigma_p, \sigma_q) \in \text{ok}\]

2. if \(\sigma_p = \sigma_q\):
\[\forall (p, q) \exists a \in a : (\sigma_a, \sigma_b) \in \text{ad}\]

3. Otherwise:
\[\forall (p, q) \exists (p, a) : ((\sigma_p, \sigma_a), (\sigma_q, \sigma_b)) \in (\text{ok}, \text{ad})\]

Cases (1) and (2) yield from the fact that core incorrectness rules of adversarial logic are the same as incorrectness logic. We shall provide additional proofs for rules *Read*, *Write*, *Success* and *Failure* which are new to AL. Case (3) is necessary when both program and adversary take steps together, as done in parallel composition, communication, backward variant and adversarial consequence rules.

**Definition 4 (Interpretation of specifications).** \([\text{ok}: p][\text{ad}: a]|(C_1 || C_2)[\text{ok}: q][\text{ad}: b]\) is true iff the adversarial triple \([p][a]|([C_1 || C_2])[\text{ok}, \text{ad}])[q][b]\) holds.

Proving that this equivalence holds for AL requires proving the soundness theorem of adversarial logic.

**Theorem 1 (Soundness).** Every adversarial logic proof is validated by the rules of adversarial denotational semantics.

To prove soundness, we appeal to the following substitution lemma generalized from reverse hoare logic [13], which we hold true without proving it.

**Lemma 2 (Substitution).** \(\sigma \in P(n/x) \iff (\sigma|x \rightarrow n) \in P\). That is:

- \(\sigma_p \in P(n/x) \iff (\sigma_p|x \rightarrow n) \in P\) if \(x \in \sigma_p\)
- \(\sigma_a \in A(n/x) \iff (\sigma_a|x \rightarrow n) \in A\) if \(x \in \sigma_a\)
The substitution lemma can be instantiated for the program relation as well as the adversarial relation when \( x \in Vars \). There is no ambiguity allowed since AL forbids variable sharing. We also follow de Vries and Koutavas [13] by managing local variables using alpha-renaming, rather than using explicit substitution like O’Hearn [2]. This changes the soundness proof for the local variable rule and the assignment rule. For all symmetric cases involving \( e \), we may give the proof one of these two cases and omit the identical proof for the other side.

**Proof.** We prove soundness for each rule of Adversarial logic. For most cases, we appeal to the characterization lemma of adversarial logic semantics to show that for all post-states of semantic triples, there is a pre-state that satisfies the adversarial precondition of the corresponding rule.

**Proof (Unit).** Assume \((\sigma_p, \sigma_a) \) skip \((\sigma_q, \sigma_b)\) and \([e: P]\) skip \([e: Q]\). Show that \( \forall (\sigma_q, \sigma_b) \in [e: Q] \exists (\sigma_q, \sigma_p) \in [e: P] \). By skip rule, \( P = Q \), so \([e: P]\) is true and \((\sigma_p, \sigma_a) = (\sigma_q, \sigma_b)\). Since \((\sigma_q, \sigma_b) \in [e: P]\) and \((\sigma_p, \sigma_a) = (\sigma_q, \sigma_b)\) then \((\sigma_p, \sigma_a) \in [e: P] \).

**Proof (Constancy).** Show that \( \forall \sigma' \in [e: Q \land F] \exists \sigma \in [e: P \land F] \text{ such that } \sigma \rightarrow \sigma' \text{ and } [e: \sigma' \in P]. \) Since \( Mod(c) \cap Free(F) = \emptyset \), \( \sigma \rightarrow \sigma' \) preserves \( F \). Therefore \( \sigma \in [e: P \land F] \).

**Proof (Assume).** Let \((\sigma_p, \sigma_a)\) assume \(B (\sigma_p, \sigma_a) \) and \([e: P]\) assume\((B) [e: P \land B]\). Show that \( \forall (\sigma_q, \sigma_b) \in [e: P \land B] \exists (\sigma_q, \sigma_a) \in [e: P] \). Since \((\sigma_p, \sigma_a) = (\sigma_q, \sigma_b)\) by assumption, \((\sigma_p, \sigma_a) \in [e: P] \).

**Proof (Rand).** Assume \((\sigma_p, \sigma_a) x = \text{rand()} \) \((\sigma_q, \sigma_b)\) with \((\sigma_q, \sigma_b) = (\sigma_p \mid x \mapsto r, \sigma_a)\). Let \((\sigma_q, \sigma_b) \in [e: P(x/x') \land x = r] \) and show that \( \forall (\sigma_q, \sigma_b) \exists (\sigma_p, \sigma_a) \in [e: P] \). Let us first cover the subcase where \( x \in \sigma_p \) and \( e = \text{ok} \). Take \((\sigma_p \mid x \mapsto n) \in [\text{ok}: P] \). By the substitution lemma, \( \sigma_p \in [\text{ok}: P(n/x)] \). By assign rule, \( \sigma_q \in [\text{ok}: P(r/x)] \). That is, \( \sigma_p \in [\text{ok}: \exists x'. P(x'/x) \land x' = r] \). The second subcase where \( x \in \sigma_a \) and \( e = \text{ad} \) can be proved similarly.

**Proof (Assign).** Take \((\sigma_p, \sigma_a) x = e (\sigma_q, \sigma_b)\) with \((\sigma_q, \sigma_b) = (\sigma_p \mid x \mapsto \{e\}_{\sigma_p}, \sigma_a)\). Let \((\sigma_q, \sigma_b) \in [e: P(x/x') \land x = e(x'/x)] \) and show that \( \forall (\sigma_q, \sigma_b) \exists (\sigma_p, \sigma_a) \in [e: P] \). Let us first cover the subcase where \( x \in \sigma_p \). Take \((\sigma_p \mid x \mapsto n) \in [\text{ok}: P] \). By the substitution lemma, \( \sigma_p \in [\text{ok}: P(n/x)] \). By assign rule, \( \sigma_q \in [\text{ok}: P([e]_{\sigma_p}(x \mapsto n/x)) \). Taking \([e]_{\sigma_p}(x \mapsto n) = m \) we obtain that \( \sigma_p \in [\text{ok}: \exists x'. P(x'/x) \land x' = m] \). The second subcase where \( x \in \sigma_a \) and \( e = \text{ad} \) can be proved similarly.

**Proof (Local).** Let us first take the case where \( x \in \sigma_q \). Show that \( \forall (\sigma_q \mid x \mapsto v, \sigma_a) \in [\text{ok}: \exists x.Q] \) there is \((\sigma_p \mid x \mapsto \{e\}_{\sigma_p}, \sigma_a) \in [\text{ok}: P] \). By the substitution lemma, \( \sigma_q \in [\text{ok}: \exists x.Q(v/x)] \), that is \( \sigma_q \in [\text{ok}: \exists x.Q] \) since \( x \) is bound. By induction hypothesis and executing backward, we obtain \( \sigma_p \in [\text{ok}: P \land x = e] \). By the substitution lemma, we have \((\sigma_p \mid x \mapsto \{e\}_{\sigma_p}) \in [\text{ok}: P(e/x)] \). Since \( x \notin Free(P) \), we conclude \((\sigma_p \mid x \mapsto \{e\}_{\sigma_p}) \in [\text{ok}: P] \). The second subcase where \( x \in \sigma_b \) can be proved similarly.
Proof (Read). We first define $\sigma' = (\sigma \mid x \mapsto v, s \mapsto l)$ and prove that for all $\sigma' \in [\epsilon : \exists x' \exists s'. P(s'/s, x'/x) \land (s = s''/v \land x = v)]$ there is $(\sigma \mid s \mapsto (l:v)) \in [\epsilon : P]$. By the substitution lemma: $\sigma \in [\epsilon : \exists x'\exists s'. P(s'/s, x'/x) \land (s = s''/v \land x = v)(v/x)(l/s)]$ That is: $\sigma \in [\epsilon : \exists x'. \exists s'. P(s'/s, x'/x) \land (l = (s''/v))]$. By rewriting $s'$, we obtain $\sigma \in [\epsilon : \exists x' P((l:v)/s, x'/x)]$. Executing read backward, we get $(\sigma \mid s \mapsto (l:v), x \mapsto x') \in [\epsilon : P]$. We can conclude since $(\sigma \mid s \mapsto (l:v), x \mapsto x') \subseteq \{ (\sigma \mid s \mapsto (l:v)) \}$

Proof (Write). Let $\llbracket \text{write}(s, x)\rrbracket \epsilon = \{(\sigma \mid s \mapsto l, x \mapsto v), (\sigma \mid s \mapsto (l:v))\}$ and $[\epsilon : P \land x = y \land s = l] \text{write}(s, x) [\epsilon : \exists s'. P(s'/s) \land s = (s''/v)]$. Define $\sigma' = (\sigma \mid s \mapsto (l:v))$ and show that $\forall \sigma' \in [\epsilon : \exists s'. P(s'/s) \land s = (s''/v)]$ there is a $(\sigma \mid s \mapsto l, x \mapsto v) \in [\epsilon : P \land x = v \land s = l]$. By the substitution lemma, $\sigma \in [\epsilon : \exists s'. P(s'/s) \land (l:v) = (s''/v)]$. That is, $\sigma \in [\epsilon : P(s'/s) \land s' = l]$. By inlining $s'$ we get $\sigma \in [\epsilon : P(l/s) \land s = (l:v)]$. By executing write backward, we obtain $\sigma \in [\epsilon : P \land x = v \land s = l]$.

Proof (Com). Assume $[\llbracket \text{Com}(C_1, C_2)\rrbracket]_{\epsilon_1, \epsilon_2} = \{((\sigma_1 \mid s \mapsto (v:l_1)), (\sigma_2 \mid s \mapsto (l_2))), (\sigma_1', \sigma_2')\}$ with $\epsilon_1, \epsilon_2 \in \{ok, ad\}$ and $\sigma_1' = (\sigma_1 \mid s \mapsto l_1)$ and $\sigma_2' = (\sigma_2 \mid s \mapsto (l_2:v))$. Prove for all $(\sigma_1', \sigma_2') \in [\epsilon_1 : \exists s'. P(s'/s) \land s = (s''/v)] [\epsilon_2 : \exists s'. A(s'/s) \land s = (s''/v)]$ there exists $(\sigma_1 \mid s \mapsto (v:l_1)), (\sigma_2 \mid s \mapsto l_2) \in [\epsilon_1 : P] [\epsilon_2 : A]$. By the substitution lemma, $(\sigma_1 \mid s \mapsto (v:l_1)), (\sigma_2 \mid s \mapsto l_2) \in [\epsilon_1 : P((v:l_1)/s)] [\epsilon_2 : A]\llbracket l_2/s \rrbracket$. Introducing $s'$, we have $((\sigma_1 \mid s \mapsto (v:l_1)), (\sigma_2 \mid s \mapsto (l_2:v)) \in [\epsilon_1 : \exists s'. P(s'/s) \land s' = (v:l_1)] [\epsilon_2 : \exists s'. A(s'/s) \land s' = (l_2:v)]$. By $[\llbracket \text{Com}(C_1, C_2)\rrbracket]$ rule, $(\sigma_1', \sigma_2') \in [\epsilon_1 : \exists s'. P(s'/s) \land s' = (v:l_1) \land s = l_1] [\epsilon_2 : \exists s'. A(s'/s) \land s' = l_2].$ Rewriting $s$ using $s'$ we now have: $(\sigma_1', \sigma_2') \in [\epsilon_1 : \exists s'. P(s'/s) \land s = (s''/v)] [\epsilon_2 : \exists s'. A(s'/s) \land s = (s''/v)]$.

Proof (Iterate). Immediate by semantic definitions and Iterate rules.

Proof (Sequencing). Immediate by semantic definition and induction hypotheses.

Proof (Choice). Immediate by semantic definition and induction hypotheses.

Proof (Disjunction). Immediate by logical definition and $\wedge \vee$ symmetry [2] of AL.

Proof (Consequence). Immediate by logical definition and $\mathcal{U}$ symmetry [2] of AL.

Proof (Par). Immediate by semantic definitions and induction hypotheses.

Proof (Success). Assume $(\sigma_p, \sigma_n) \text{ adv \_assert}(B) \{((\sigma_q, \sigma_n) \mid \llbracket B \rrbracket_{\sigma_n} = true\}$ by $\mathcal{U}$ left subset. Success rule gives us that $[\text{ad} : P \land (P \Rightarrow B)] \text{ adv \_assert}(B) [\text{ad} : P \land true]$. Show that $\forall (\sigma_q, \sigma_n) \in P \land true \exists (\sigma_p, \sigma_n) \in P \land (P \Rightarrow B)$. Success rule does not modify any variable of $(\sigma_p, \sigma_n)$, therefore $(\sigma_p, \sigma_n) = (\sigma_q, \sigma_n)$ and $(\sigma_p, \sigma_n) \in P \land B$. Since $(P \land B) \iff (P \Rightarrow B)$, we conclude that $(\sigma_p, \sigma_n) \in P \land (P \Rightarrow B)$.  

Relational Adversarial Logic 21
Proof (Failure). Assume \((\sigma_p, \sigma_a)\) \texttt{adv\_assert}(B) \{\{(\sigma_q, \sigma_b) \mid \llbracket B \rrbracket_{\sigma_a} = \text{false}\}\} by \(\cup\) right subset. Failure rule gives us that \([\text{ad}: P \land (P \Rightarrow \neg B)]\) \texttt{adv\_assert}(B) \([\text{ad}: P \land \neg B]\). Show that \(\forall(\sigma_q, \sigma_b) \in P \land \neg B \exists(\sigma_p, \sigma_a) \in P \land (P \Rightarrow \neg B)\). Failure rule does not modify any variable of \((\sigma_p, \sigma_a)\), therefore \((\sigma_p, \sigma_a) = (\sigma_q, \sigma_b)\) and \((\sigma_p, \sigma_a) \in P \land \neg B\). Since \((P \land \neg B) \iff (P \Rightarrow \neg B)\), we conclude that \((\sigma_p, \sigma_a) \in P \land (P \Rightarrow \neg B)\).

Proof (Adversarial Consequence). We know that \((A' \land \exists v.1.Q \land v1 = v2) \Rightarrow A'\). Applying the consequence rule backwards, \(\sigma_b \in [\text{ad}: A' \land \exists v.1.Q \land v1 = v2]\) implies \(\forall \sigma_q \in [\text{ad}: A']\). Therefore by induction hypothesis, we know \(\exists \sigma_a \in [\text{ad}: A]\). By the second induction hypothesis, we also know that \(\forall \sigma_q \in [\text{ok}: P'] \exists \sigma_p \in [\text{ok}: P]\). Applying the parallel rule backward, we obtain that \(\forall(\sigma_q, \sigma_b) \in [\text{ok}: P'][\text{ad}: A'] \exists(\sigma_p, \sigma_a) \in [\text{ok}: P][\text{ad}: A]\).

Proof (Parallel Backward Variant). We show that \(\forall(\sigma_q, \sigma_b) \in [\text{ok}: \exists n. P(n)][\text{ad}: \exists m. A(m)]\) there exists \((\sigma_p, \sigma_a)\) such as \((\sigma_p, \sigma_a) \rightarrow (\sigma_q, \sigma_b)\) and \((\sigma_p, \sigma_a) \in [\text{ok}: P(0)][\text{ad}: A(0)]\).

Proof (Case \(n = m = 0\)). Immediate by definition of \texttt{Iterate Zero} rule, with \((\sigma_p, \sigma_a) = (\sigma_q, \sigma_b)\).

Proof (Case \(n = m\) and \(i = j = 1\)). By inductive hypothesis, it holds that \([\text{ok}: P(n-1)][\text{ad}: A(m-1)]\&\exists c_1\|c_2\|\texttt{ok}: P(n)][\text{ad}: A(m)]\) and there is a \((\sigma_p(n-1), \sigma_a(m-1))\) \(\in [\text{ok}: P(n-1)][\text{ad}: A(m-1)]\). We reuse the induction hypothesis several times going backward until we reach \((\sigma_p, \sigma_a) \in [\text{ok}: P(0)][\text{ad}: A(0)]\).

Proof (Case \(n \neq m\)). By inductive hypothesis, it holds that \([\text{ok}: P(n)][\text{ad}: A(m)]\&\exists c_1\|c_2\|\texttt{ok}: P(n+i)][\text{ad}: A(m+j)]\). Therefore, \(\exists(\sigma_q(n+i), \sigma_b(m+j)) \in [\text{ok}: P(n+i)][\text{ad}: A(m+j)]\). Define \(\delta(n, m) : (\mathbb{N} \times \mathbb{N}) \rightarrow (\mathbb{B} \times \mathbb{B})\) the function mapping values of \((n, m)\) to their corresponding values \((i, j)\) where \(i, j \in \{0, 1\}\). We have three subcases:

- \(\delta(n, m) = (0, 1)\) and \(\exists(\sigma_q(n), \sigma_b(m-1)) \in [\text{ok}: P(n)][\text{ad}: A(m-1)]\).
- \(\delta(n, m) = (1, 0)\) and \(\exists(\sigma_q(n-1), \sigma_b(m)) \in [\text{ok}: P(n-1)][\text{ad}: A(m)]\).
- \(\delta(n, m) = (1, 1)\) and \(\exists(\sigma_q(n-1), \sigma_b(m-1)) \in [\text{ok}: P(n-1)][\text{ad}: A(m-1)]\).

Recursively going backward using one of the three subcases, we eventually reach one of the two following termination conditions:

- The program reaches its initial condition before the adversary:
  - \((\sigma_q(0), \sigma_b(m-j)) \in [\text{ok}: P(0)][\text{ad}: A(m-j)]\).
  - For all remaining \((m - j)\) steps, we have \(\delta(0, m) = (0, 1)\)
  - \((\sigma_{p0}, \sigma_{a0}) \xrightarrow{n-j} (\sigma_{p0}, \sigma_{b(m-j)}) \in [\text{ok}: P(0)][\text{ad}: A(m-j)]\)

- The adversary reaches its initial condition before the program:
  - \((\sigma_{q(n-i)}, \sigma_{b0}) \in [\text{ok}: P(n-i)][\text{ad}: A(0)]\).
  - For all remaining \(n - i\) steps, we have \(\delta(n, 0) = (1, 0)\)
  - \((\sigma_{p0}, \sigma_{a0}) \xrightarrow{n-i} (\sigma_{p(n-i)}, \sigma_{b0}) \in [\text{ok}: P(n-i)][\text{ad}: A(0)]\)
7 Discussion And Related Work

Adversarial Logic extends Incorrectness Logic by O’Hearn [2]. We wish to stress that AL (just like IL) is a forward logic. This can be seen in individual rules such as the `read`, `write` and `assign` rules which use variants of Floyd’s axiom rather than backwards substitution like in Hoare logic. As seen in section 6, the characterization lemma of AL is a straight translation from IL extended to the parallel setting where the adversarial program and the target program are composed using `||`. Several programs can also be composed with the adversary as needed for the relational analysis.

Concurrent Incorrectness Reasoning has been pioneered by Raad et al in Concurrent Incorrectness Separation Logic [11] in which race conditions can be formalized. In AL, the parallel composition is used as a logic harness rather than to model an actual concurrent execution of threads sharing state. One key difference between AL and IL and its descendent is the elimination of short-circuiting rules.

Table 8: Differences between characterization lemmas of Incorrectness Logic [2], Adversarial Logic [1] and Insecurity IL [15].

<table>
<thead>
<tr>
<th>Judgments</th>
<th>Characterization Lemma</th>
</tr>
</thead>
<tbody>
<tr>
<td>IL [c: P] \ [c: Q]</td>
<td>[\forall \sigma_q \in [c: Q] \exists \sigma_p \in [c: P] ]</td>
</tr>
<tr>
<td>AL [ok: P]</td>
<td>[ad: A] \ \ [ok: Q]</td>
</tr>
<tr>
<td>InsecIL [\prec R &gt; s, s' &lt; S &gt;]</td>
<td>[\forall t t'. S(t, t') \Rightarrow (\exists s s'. R(s, s') \land (c, s) \Rightarrow t \land (c', s') \Rightarrow t')]</td>
</tr>
</tbody>
</table>

Relational analysis predates recent efforts in underapproximate program logic. Naumann references no less than 37 years of research in relational Hoare Logic [16], where properties such as determinacy, non-interference, equivalence, mutual termination and others are defined. None of such properties are reachable for an under-approximate program logic, which does not reason on all program paths, but rather a subset of them. Unlike those properties documented by Naumann, relational properties described in the past few sections pertain to agreement and disagreement e.g. that programs agree on the same output for a given input, or that they differ for a chosen (set of) input. As such, no inductive invariant or lock-step behavior is assumed from the programs under analysis, making under-approximate analysis a versatile framework providing a sound theoretical underpinning for bug finding and exploitability.

A relational under-approximate logic was presented by Murray et al. under the name Insecurity Logic [15]. Differences between the interpretation of each logic can be seen in table 8. The inheritance from IL to AL is flagrant, while InsecIL stands separated. There are several consequences of this difference for InsecIL. First, binary relations \(R\) and \(S\) are limiting the relational analysis to
two programs, while the encoding of comparable programs in AL can be made by
parallelly composing whatever desired number of programs. The second differ-
ence (and not the least) is that InsecIL soundness cannot be proved by extending
the soundness result from IL, unlike AL which is a parallel extension of IL. Some
of the details of the InsecIL soundness proof are not published, which makes it
difficult to precisely compare it with the detailed soundness proof available for
AL in section 6.

Incorrectness Separation Logic [10] is the analogous of Separation Logic [9]
in the under-approximate setting, allowing to reason on pointer programs with
dynamic memory allocation. Adding adversarial reasoning to CISL is a natural
extension of AL so that it can handle checking properties of pointer-manipulating
programs. In this article, we focused on properties other than memory safety,
therefore not strictly requiring such separation axioms.

Adversarial logic draws inspiration from the spi-calculus [17], a dialect of
the pi-calculus [18] designed for cryptographic protocol verification of secrecy
properties. The spi-calculus requires an abstract model representation of such
protocols, rather than an actual program implementation like AL. Villard et
al. also described a program logic for heap reasoning where pointer ownership
is passed between threads communicating via messages, dubbed heap hop [19].
AL is an under-approximate flavor program logic which shares many conceptual
characteristics with these two over-approximate formal systems.

Recently, a number of new program logics have been defined to reason about
correctness and incorrectness within a unified framework. Outcome logic [20]
(OL) and Hyper-Hoare Logic [21] (HHL) allow to express verification conditions
as well as incorrectness conditions through refutation of specifications. HHL
is designed to verify hyper-properties, unlike OL which currently has no such
application. OL may nonetheless be extended with relational capabilities in the
future. Neither OL nor HHL support the parallel composition of programs as
employed in (R)AL [1] and CISL [11].

8 Conclusion

We presented Relational Adversarial Logic (RAL), a program logic derived from
Adversarial Logic [1] dedicated to the relational analysis of security properties.
We demonstrated the usage of RAL on a real-world URL parsing confusion vul-
nerness where two implementations of URL parsing implementing different
standards allows an attacker to confuse clients to make them visit undesired
addresses. This interpretation of relational analysis strikes as pragmatic and re-
alistically can tackle complex examples of security issues that are due to behavior
divergence between comparable programs.
References