Adversarial Logic

Julien Vanegue(✉)

Bloomberg, New York, USA
jvanegue@bloomberg.net

Abstract. We introduce *Adversarial Logic*, an extension of Incorrectness Logic [1] with an explicit Dolev-Yao [2] adversary to statically analyze the severity of security vulnerabilities in the under-approximate setting. Adversarial logic is built on the ability to separate logical facts known to the adversary from facts solely known to the program under analysis. This flavor of program incorrectness can be used to analyze software in which error behavior occurs at deeper levels of interaction between the program and its environment, such as subtle cases of information disclosure requiring multiple program executions to be uncovered. We introduce the *Oscillating Bit Protocol*, an example algorithm where such a vulnerability can be detected using adversarial logic while remaining elusive to other frameworks. We define a flavor of symbolic execution in which the adversary guides the introduction of symbolic variables and the checking of attack assertions. Additionally, we introduce *equivalence testing*, an under-approximate version of program equivalence only proven on specific program paths and used to extract differences between comparable implementations. We provide a denotational semantics for adversarial logic and prove its soundness, thereby guaranteeing that extracted attack paths are true positives.

1 Introduction

The ever growing volume of software developed over the last several decades has led to a situation where software vendors and open source projects are unable to keep up with the sheer number of vulnerabilities found and disclosed by the community every year. In just the Linux kernel, a single testing tool [3] identified more than 3,000 bugs in two years, and thousands of others are regularly found in mainstream projects [4]. A crucial problem in the presence of such large numbers of bugs is to determine the practical security implications of each bug to inform which bugs must get fixed first.

Particularly dangerous software attacks attempt to elevate privileges [5] or steal secrets [6] using an adversarial program (or *exploit*) manually written by a security expert. Determining the potential for compromise of a vulnerable program is a time-consuming task of paramount importance, that has received surprisingly little attention from the formal verification community. This is especially critical as (a) known bugs are left unremediated for a long time, (b) security compromises are increasingly costly and (c) existing tools keep finding hundreds of new bugs every month.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
G. Singh and C. Urban (Eds.): SAS 2022, LNCS 13790, pp. 422–448, 2022.
https://doi.org/10.1007/978-3-031-22308-2_19

This paper provides a logical foundation for *exploit programming* [7] dedicated to the static *exploitability* analysis of program bugs. Not all bugs are considered equal from an exploitability perspective: Can it be used to divert the program's control flow or corrupt data [8]? Does it allow information disclosure leading to password or private key compromise [6]? Does it allow untrusted code execution as root, or other remote user [5]? Elaborate exploits often require multiple stages of probing the target for reconnaissance [9] or leverage several weak bugs to build a complete attack. For example, an exploit may first attempt to guess internal program addresses using an information disclosure vulnerability to infer the location of sensitive data [10], and then use a subsequent array out of bound access to tamper with said data. Analyzing such bug chaining currently remains out of reach for existing program analysis frameworks.

Historically, program verification has focused on sound and over-approximate analysis guaranteeing the absence of entire classes of bugs in analyzed software [11,12] at the expense of false positives. To work around program analysis undecidability, recent trends are focusing on under-approximate and complete analyses in which findings are guaranteed to uncover real issues. Theoretical underpinning of bug finding now enjoys a foundational theory of incorrectness logic [1] (IL), a new logic focusing on uncovering true bugs rather than proving the absence of bugs. Fuzz testers [3,4] and other complete tools are indeed immediately actionable in the software development life-cycle of large software organizations, where the absence of false positives is critical to developer adoption. Incorrectness logic was recently extended to include heap reasoning [13] and concurrency checking [14] demonstrating its versatile nature. Adding an explicit attack program to reason about adversarial behaviors of incorrectness is a fundamentally under-approximate problem, and naturally extends IL.

We introduce adversarial logic (AL), a new under-approximate logic extending incorrectness logic (IL) to determine bug exploitability by leveraging accumulated error in software programs. Adversarial reasoning can be used as a theoretical basis to determine the existence of true attacks in buggy software. The resulting logic gives rise to a notion of *attack soundness* that captures sufficient conditions for an attack to be guaranteed satisfiable by an adversary. To prove soundness, it is sufficient to demonstrate that some execution paths of the program are exploitable. This differs from typical verification frameworks which attempt to prove statements about all possible program paths. We prove the main attack soundness result of this paper in Sect. 5. AL extends IL in the following ways:

- We consider the system under analysis to be a parallel composition of the analyzed program and an explicit adversarial program attempting to falsify the program specification.
- We focus on proving the satisfiability of an *attack contract* rather than following the usual methodology of checking code contracts in the program itself.
- We introduce adversarial preconditions, which allows for program errors to accumulate transitively. IL has no error preconditions and only error postconditions.

- We add channel communication rules to model that only explicitly shared program output is visible to an external adversary.
- We introduce a new *adversarial consequence* rule to derive additional adversarial knowledge otherwise remaining unobserved in program output.
- We generalize the backward variant rule of IL to the parallel case, so that AL can determine the existence of attacks in interactive protocol loops without unrolling the entire attack path.

The new adversarial consequence rule is of particular importance to the discovery of indirect information disclosure attacks, as AL can model leaking of internal program state without assuming its direct observability by the adversary.

It is worth noting that adversarial logic does not encode *root cause analysis* [15] as it does not attempt winding back to the source of bugs. Rather, it provides a framework to analyze *bug effects* by considering unintended computations as first-class primitives, allowing the transitive tracking of error conditions through the adversarial interpretation of the program.

The rest of the paper is organized as follow: we introduce a motivating example of an information disclosure attack in the *Oscillating Bit Protocol* in Sect. 2. We introduce the rules of Adversarial Logic in Sect. 3. We explore additional examples demonstrating the usage of AL in Sect. 4. Among these new examples is a technique to under-approximate program equivalence we call *equivalence testing*. We give the denotational semantics of AL and prove soundness of AL rules with respect to its semantics in Sect. 5. We briefly provide alternative presentations to adversarial logic based on the formalism of dynamic logic [16] and information systems [17] in Sect. 6. Finally, we cover related work in Sect. 7 and conclude on future work.

2 Motivation

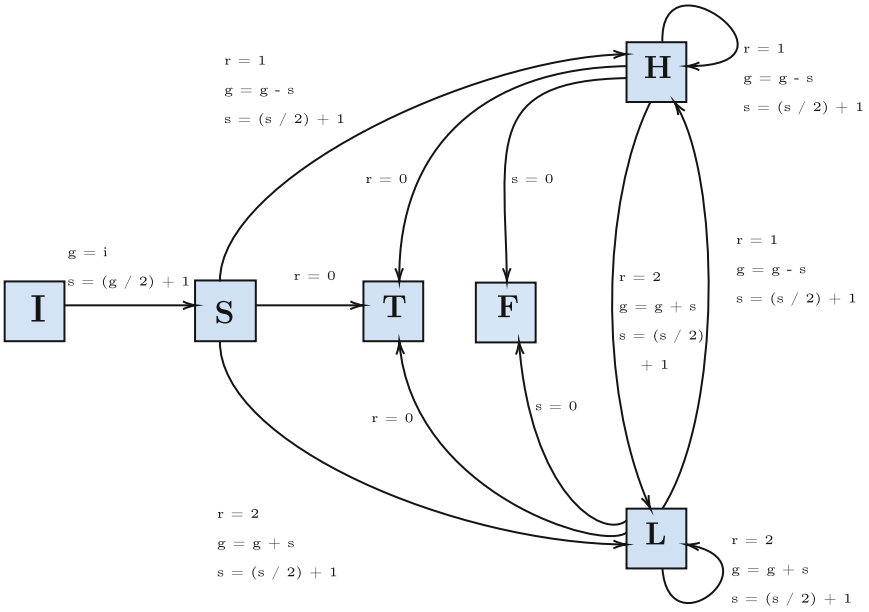
Let us start with an example in Table 1 where an information disclosure attack is performed in $\mathcal{O}(n)$ interactions with the program. Note how the value of the *secret* variable is used to grant access to the function *do_serve*. Due to a discrepancy in the return value of the *server* function, it is possible for an adversary to determine the secret without reading it directly. The server's observable return value will be 0 (the adversary's goal encoded in *adv_assert* on line 14), or 1 (the provided value was too big) or 2 (the provided value was too small). Therefore, an adaptive search can guess the secret value in a maximum attempts of $\mathcal{O}(n)$ instead of the naive brute-force algorithm in $\mathcal{O}(2^n)$ where n is the size of the secret in bits. The oscillating nature of checking the adversarial assertion is represented as a finite state automaton in Table 2.

Table 1. Oscillating bit protocol: target program (Left) and adversary (Right)

```
// pre: client socket established
1. uint8 secret = rand8();
2.
3. void server(int sock)
4. {
5.  uint8  err = 0 in
6.  uint8  cred = 0 in
7.  while (true) do
8.    read(sock, cred);
9.     if (secret == cred)
10.        err = 0;
11.    else if (secret < cred)
12.        err = 1;
13.    else if (secret > cred)
14.        err = 2;
15.    if (!err) do_serve(sock);
16.    write(sock, err);
17.  done
18.}
```

```
// pre: socket established to server
1. int client(int sock)
2. {
3.  uint8 ret = 1 in
4.  uint8 guess = UINT8_MAX in
5.  uint8 step =  (UINT8_MAX / 2) + 1 in
6.  while (true) do
7.     write(sock, guess);
8.     read(sock, ret);
9.     if (ret == 1)
10.      guess = guess - step;
11.   else if (ret == 2)
12.      guess = guess + step;
13.   step = (step / 2) + 1;
14.   adv_assert(ret == 0);
15.  done
16.}
```

Table 2. OBT attack has initial state (I) and started state (S) then oscillates between high (H) and low (L) before terminating in success (T) or failure (F).



Since integer are represented with 8-bits in this example, the adversary would need to consider only $8 + 1 = 9$ values before it can successfully guess the secret and satisfy the adversarial assertion. Note that width 8 is chosen for simplicity, and this class of *linear-complexity attacks* scales very well when x grows to 16, 32, 64, etc. Not all attacks are that simple in practice, and one can imagine polynomial or more complex attack strategies up to infinite ones. The full sequence of interactions in the oscillating bit protocol is given in appendix.

3 Adversarial Logic

Adversarial Logic (AL) marries under-approximate reasoning [1,18] with an adversarial model [2] suitable for the study of complex software implementation attacks. We will work with this toy imperative language made of variables, expressions, channels, predicates and commands.

Variables	V ::= x | n | α
Expressions	E ::= V | rand() | E + E | E - E | ...
Channels	L ::= s | $\emptyset$ | (V::L) | (L::V) | (s \ 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 || 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)

Expressions are made of named variables x, concrete integers n, symbolic values α, 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\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.

For simplicity, a small set of scalar variable types T is available, which is sufficient to cover all examples of 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 *rand()* value as an *uint8* (such as the oscillating bit protocol in Table 5) and other times as a *float* (as in the equivalence testing example of Table 7). We will adopt *rand8()* or *randf()* as needed to make precise which version is used, or simply *rand()* when the version is obvious from context.

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 *Com* is distinct from read and write, and *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 caching strategy. Although studies of specific caching strategies are out of scope for this paper, attacks leveraging caching behavior are now mainstream [19,20] and it is critical to allow a spectrum of possibilities as to when communication effectively happens on channels. As such, we leave *Com* implicit in our examples and apply the corresponding proof rule when required to make progress.

Although AL can reason about the full adversarial term when available, it is not required to be the case and a minimal template is often sufficient. Example 2 and 3 of Sect. 4 show how such templates can be leveraged to build attack proofs. Additional syntactic sugar is defined for convenience purpose:

$$\text{if } P\ c \stackrel{\text{def}}{=} \text{if } P \text{ then } c \text{ else } \mathit{skip}$$
$$\mathrm{T}\ x = E_1, y = E_2 \text{ in } C \stackrel{\text{def}}{=} \mathrm{T}\ x = E_1 \text{ in } (\mathrm{T}\ y = E_2 \text{ in } C)$$
$$\text{if } P\ c_1 \text{ else if } Q\ c_2 \text{ else } c_3 \stackrel{\text{def}}{=} \text{if } P \text{ then } c_1 \text{ else } (\text{if } Q \text{ then } c_2 \text{ else } c_3)$$

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

$$[ok\colon P]\ \mathrm{c}\ [ok\colon Q][er\colon R]$$
$$\stackrel{\text{def}}{=}$$
$$[ok\colon P]c[ok\colon Q] \text{ and } [ok\colon P]c[er\colon R].$$

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

$$[ok\colon P_1][ad\colon P_2]\ \mathrm{c}\ [ok\colon Q_1][ad\colon Q_2]$$
$$\stackrel{\text{def}}{=}$$
$$[ok\colon P_1]c[ok\colon Q_1] \text{ and } [ad\colon P_2]c[ad\colon Q_2].$$

Program interpretation and adversarial interpretation are compositional and allow independent reasoning over ok and 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.

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

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

More succinctly, rules in AL are written as:

$$[\epsilon\colon P]\ \mathrm{c}\ [\epsilon\colon Q]$$

where $\epsilon \in \{ok, ad\}$ is a short notation to write two rules $[ok\colon P]$ c $[ok\colon Q]$ and $[ad\colon P]$ c $[ad\colon Q]$ 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.

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

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

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 c_p and an adversary c_a is constructed from a program interpretation and an adversarial interpretation of parallel terms:

$$[ok\colon P_1][ad\colon P_2]\ c_p\ ||\ c_a\ [ok\colon Q_1][ad\colon Q_2]$$

Two parallel terms may either be an adversarial term and a program term, or two independent program terms, with ϵ_1,$\epsilon_2 \in \{ok, ad\}$:

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

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 [21].

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.

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

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 Sect. 4.

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

Adversarial logic generalizes the **Backward variant** rule of incorrectness logic [1] for parallel composition of program and adversarial terms, which we name the Parallel Backward Variant, or **PBV**.

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

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 ($i, j \in \{0, 1\} \wedge i + j \geq 1$). This condition enforces that every extracted attack trace is finite. Examples in this article show cases where $n = m$ (the oscillating bit protocol), and others where $n \neq m$ (equivalence testing).

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 5 of Sect. 4.

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 *er* 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.

(a) Adversarial logic core rules with restored symmetry from IL : $\epsilon \in \{ok, ad\}$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4 Reasoning with Adversarial Logic

In this section, we put AL to work with three distinct examples. The simplest example of Table 3 is sufficient to explain symbolic variable introduction, adversarial consequence and assertion checking by the adversary. The oscillating bit protocol example from Table 1 of Sect. 2 is then proved step-by-step, including a proof showing the use of the parallel backward variant (PBV) rule for the determination of the existence of attacks. In example 3, two pricing functions are under-approximated to find common price boundaries through adversarial assertions and combines usage of the PBV rule and the adversarial consequence rule to perform equivalence testing.

4.1 Example 1: Trivial Case

Let us consider the example in Table 3 where an adversary wants to capture a flag *win* with an input value n reaching value 10 million (10M for short).

Table 3. Simple example implementation. The adversary wishes to discover conditions on symbolic variable *val* to satisfy assertion (res == 1)

```
// 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 channel established
adversary(int s)
{
  uint32 val = α in
  uint32 res = 0 in
  write(s, val);
  read(s, res);
  adv_assert(res == 1);
}
```

An adversarial proof such as the one in Table 4 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_3 and A_0 to A_3) where program and adversary have yet to be composed.
2. The initial phase (P_3, A_3) to (P_9, A_9) typically starts with application of the **Par** rule until composed terms fail to make more progress other than **Skip**
3. Optionally, one or more intermediate phases separated by applications of the **Com** rule used to communicate and unblock stuck terms, interleaved with calls to **AdvAssert** failing to satisfy the attack contract.
4. The final phase ends with a call to **Success** where the adversarial assertion is satisfied (A_{12}), or when the adversarial program terminate otherwise.

Table 4. Simplest example in Adversarial Logic. Unlike traditional program symbolic execution, assertions and symbolic variables can only be introduced in the adversarial part of the system.

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

$\rightarrow Par$	$(P_3, A_3) = \{ok\colon P_3\}\{ad\colon A_3\}$
$\rightarrow Com$	read(s,n) || read(s,res)
	$(P_4, A_4) = \{ok\colon \exists \alpha \exists s_p^2.P_3(s_p^2/s_p) \wedge s_p = (s_p^2::\alpha)\}\{ad\colon \exists s_a^2.A_3(s_a^2/s_a) \wedge s_a = (s_a^2 \backslash \alpha)\}$
$\rightarrow Read$	read(s,n) || read(s, res)
	$(P_5, A_5) = \{ok\colon \exists \alpha \exists s_p^3 \exists n_2.P_4(s_p^3/s_p, n_2/n) \wedge s_p = (s_p^3 \backslash \alpha) \wedge n = \alpha\}\{ad\colon A_4\}$
$\rightarrow If, Assn$	if (n > 10M) win = 1 || read(s, res)
	$(P_6, A_6) = \{ok\colon \exists w_2.P_5(w_2/win) \wedge n > 10M \wedge win = 1\}\{ad\colon A_5\}$
$\rightarrow If, Assn$	else win = 0 || read(s, res)
	$(P_7, A_7) = \{ok\colon \exists w_3.P_5(w_3/win) \wedge n \leq 10M \wedge win = 0\}\{ad\colon A_6\}$
$\rightarrow Disj$	$(P_8, A_8) = \{ok\colon P_6 \vee P_7\}\{ad\colon A_7\}$
$\rightarrow Write$	write(s, win) || read(s, res)
	$(P_9, A_9) = \{ok\colon \exists w \exists s_p^4.P_8(s_p^4/s_p) \wedge s_p = (s_p^4::w) \wedge win = w\}\{ad\colon A_8\}$
$\rightarrow Com$	skip || read(s,res)
	$(P_{10}, A_{10}) = \{ok\colon \exists s_p^5.P_9(s_p^5/s_p) \wedge s_p = (s_p^5 \backslash w)\}$ $\{ad\colon \exists w \exists s_a^3.A_9(s_a^3/s_a) \wedge s_a = (s_a^3::w)\}$
$\rightarrow Read$	skip || read(s, res)
	$(P_{11}, A_{11}) = \{ok\colon P_{10}\}\{ad\colon \exists s_a^4 \exists r_2 \exists w.A_{10}(s_a^4/s_a, r_2/res) \wedge s_a = (s_a^4 \backslash w) \wedge res = w\}$
$\rightarrow Adv.C.$	skip || adv_assert(res == 1)
	$(P_{12}, A_{12}) = \{ok\colon P_{11}\}\{ad\colon A_{11} \wedge \exists n \exists x.((n > 10M \wedge x = 1) \vee (n \leq 10M \wedge x = 0))$ $\wedge\ n = \alpha \wedge res = x\}$
$\rightarrow Success$	skip || adv_assert(res == 1)

Note how symbolic variable α is introduced by the adversarial interpretation (A_1) and propagated to the program's logic (P_4) using the communication rule. Program and adversarial interpretations remain independent until the parallel rule is used to compose terms. Note how P_9 in insufficient to prove the assertion $res == 1$ thus requiring the application of the adversarial consequence rule to obtain additional knowledge $(n = \alpha) \wedge (n > 10M)$.

4.2 Example 2: Oscillating Bit Protocol

We now analyze the motivating example presented in Sect. 2. It is possible to prove existence of an information disclosure attack in the oscillating bit protocol with or without the parallel backward variant rule. As we will show, use of the PBV rule allows to significantly shorten the proof. Without it, the adversarial interpretation goes through several instances of adversarial failures where $adv\_assert(retcode == 0)$ cannot be satisfied. After a sufficient number of guesses are performed and constraints over the secret are learned, the adversary finally provides a value that matches the secret. In the OBP example, $cred == 160$ is the secret value which cannot be inferred without performing $\mathcal{O}(n)$ steps.

Table 5. Oscillating Bit Protocol (OBT) in AL using disjunction and parallel backward variant rules. Introduction of the PBV rule cut the number of needed steps and can be used to deduce the existence of an attack without guessing the secret.

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

$\rightarrow Par$	$(P_7, A_7) = \{ok\colon P_6\}\{ad\colon A_6\}$
$\rightarrow Com$	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)
	$(P_{11}, A_{11}) = \{ok\colon P_{10} \vee (secret < cred \wedge \exists e.P_{10}(e/err) \wedge err = 1)\}\{ad\colon A_{10}\}$
$\rightarrow If, Fra.$	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)
	$(P_{13}, A_{13}) = \{ok\colon \exists e \exists s_p^3.P_{12}(s_p^3/s_p) \wedge err = e \wedge s_p = (s_p^3::e)\}\{ad\colon A_{12}\}$
$\rightarrow Com$	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
	$(P_{17}, A_{17}) = \{ok\colon P_{16}\}$
	$\{ad\colon (ret = 2 \wedge \exists g.A_{16}(g/guess) \wedge guess = g + step) \vee (ret \neq 2 \wedge A_{16})\}$
$\rightarrow Assn$	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) || attack_assert(ret == 0)
	$(P_{19}, A_{19}) = \{ok\colon P_{18}\}\{ad\colon A_{18} \wedge ret \neq 0)\}$
$\rightarrow PBV$	$(P_{20}, A_{20}) = \{ok\colon \exists n.P_n : (secret = cred) \wedge (err = 0)\}\{ad\colon \exists n.A_n : (ret = 0)\}$
$\rightarrow Succ$	read(sock, cred) || attack_assert(ret == 0)

For brevity, we provide analysis of the example using PBV rule in Table 5, while the version without PBV is given in appendix. A combination of PBV and disjunction rules allows the adversary to find an iteration where the secret is correctly guessed without executing the loop $\mathcal{O}(n)$ times. Recall the form of the PBV rule with c_p the program term and c_a the adversarial term:

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

In the oscillating bit protocol example, the PBV rule takes a simpler form where $n = m$ and $i = j = 1$ for all values of (n, m). Applying the PBV rule for this example proceeds as such: $P(0)$ is $secret = v_1$, $A(0)$ is $guess = v_2$, $P(n)$ is $secret = v_1$, $A(n)$ is $guess = v_1$, $P(n+1)$ is $secret = cred$ and $A(n+1)$ is $ret = 0$. Note how $P(0)$ and $P(n)$ are both $s == v_1$ as the secret value does not change across iterations. This condition is not strictly required and may not be guaranteed in more complex examples, such as if the secret variable value changes over time.

$$\frac{[ok\colon secret = v_1][ad\colon guess = v_1]\ c_p \mid\mid c_a\ [ok\colon secret = cred][ad\colon ret = 0]}{[ok\colon secret = v_1][ad\colon guess = v_2]\ c_p^n \mid\mid c_a^n\ [ok\colon \exists P_n.secret = v_1][ad\colon \exists A_n.guess = v_1]}$$

We distinguish adversarial proofs which do not appeal to the parallel backward variant rule from those using PBV since the use of PBV allows to reach adversarial success without guessing the secret. Adversarial proofs with PBV are not sufficient to build a concrete attack, but they are sufficient to prove that an attack exists.

4.3 Example 3: Equivalence Testing

Equivalence properties are relevant to security to prove compatibility or indistinguishability of two programs. For example, comparing multiple parsing implementations of a given input language (network headers, ASN.1, etc.) can uncover subtle program behaviors allowing exploitation or fingerprinting of systems [22].

Equivalence results are generally established by showing that the labeled transition system of a program implementation is equivalent to the LTS of its specification [23]. Bisimulation requires two LTS to be observationally equivalent for all transitions. Proving such equivalence is out of reach in the under-approximate framework, in which only some program executions must be analyzed. Take for example the *epsilon-delta* definition of the limit of a function:

$$\forall \epsilon > 0 \exists \delta > 0 : |x - c| < \delta \implies |f(x) - L| < \epsilon$$

Universary quantified propositions like this one are unprovable in IL and AL, which restricts us to under-approximate equivalence testing for certain inputs. This is useful to prove that two programs are equivalent *sometimes*, and find values for which programs agree. For f_1 and f_2 : $\exists x : f_1(x) = f_2(x)$. Let us assume that $f_1(x)$ and $f_2(x)$ can be written as:

$$f_1(x) = \exists x.(P_1(x) \Rightarrow Q_1(x)) \wedge ... \wedge (P_n(x) \Rightarrow Q_n(x))$$
$$f_2(x) = \exists x.(R_1(x) \Rightarrow T_1(x)) \wedge ... \wedge (R_m(x) \Rightarrow T_m(x))$$

We use this general form where P_1 to P_n (resp. R_1 to R_m) represent the path conditions associated to output values Q_1 to Q_n (resp. T_1 to T_m). Existence of a crossing point between f_1 and f_2 can now be written as:

$$EquTst(f_1, f_2) = \exists x \exists i \exists j : (P_i(x) \Rightarrow Q_i(x)) \wedge (R_j(x) \Rightarrow T_j(x)) \wedge (Q_i(x) \Leftrightarrow T_j(x))$$

Testing equivalence of f_1 and f_2 is computable in adversarial logic even when internal program variables (possibly random ones) are involved in the calculation of f_1 or f_2. Equivalence testing does not require proving $P_i \Leftrightarrow R_j$ (as required in bisimulation) as internal computations P_i and R_j may be hidden to the adversary. Hence equivalence testing is neither a bisimulation nor a simulation.

Table 6. Two pricing functions with user-supplied order number and random initial price.

```
                                  // Client: Adversarial software
                                  // Precond: chan s1 and s2 established
                                  1. void      Adv(int s1, int s2)
                                  2. {
// Preprocessor definitions       3.    float  guess = 0 in
 1. define V1MIL 1000000          4.    float  guess2 = 0 in
 2. define V9MIL (V1MIL*9)        5.    uint32 num = α in // Sym
 3. define V18MIL (V1MIL*18)      6.    write(s1, num); // Test GP1
 4. define V10MIL (V1MIL*10)      7.    read(s1, guess1);
 5. define V20MIL (V1MIL*20)      8.    write(s2, num); // Test GP2
 // Shared initial service state  9.    read(s2, guess2);
 // Between GetPrice and GetPrice2 10.  adv_assert(guess1 == guess2);
 6. float initp = rand();         11. }

// Precond: chan s1 established  // Precond: channel s2 established
 7. void GetPrice(int s1)        22. void GetPrice2(int s2)
 8. {                            23. {
 9.   float curp in               24.   float curp2 in
10.   uint32 ord in              25.   uint32 ord2 in
11.   float dec in               26.   float dec2 in
12.   while (true) do            27.   while (true) do
13.     read(s1, ord);           28.     read(s2, ord2);
14      dec = ord / V10MIL;      29.     dec2 = ord2 / V20MIL;
15.     if (ord <= V9MIL)        30.     if (ord2 <= V18MIL)
16.       curp = initp * (1 - dec); 31.       curp2 = initp * (1 - dec2);
17.     else                     32.     else
18.       curp = initp / 10;     33.       curp2 = initp / 10;
19.     write(s1, curp);         34.     write(s2, curp2);
20.   done                       35.   done
21. }                            36. }
```

Consider the code in Table 6 where a pricing service contains two functions *GetPrice* and *GetPrice2* reading on channels s_1 and s_2 to compute market price based on a globally initialized random market value *initp* and ordered quantities *num*. The first function converges faster than the other due to a different current price calculation.

Adversarial logic can be used to prove that functions *GetPrice* and *GetPrice2* meet at the same limit price (a tenth of the initial price) for certain input order

Table 7. Equivalence testing: PBV and adversarial consequence rules are combined to find an input for which two pricing functions have the same output.

$\rightarrow Rand$	$P_0 = \{ok\colon \emptyset\}$ float initp = rand(); $P_1 = \{ok\colon \exists f.initp = f \wedge P_0\}$	$A_0 = \{ad\colon \emptyset\}$ float guess1; $A_1 = \{ad\colon \exists v_1.guess1 = v_1 \wedge A_0\}$ $\rightarrow Loc$ float guess2; $A_2 = \{ad\colon \exists v_2.guess2 = v_2 \wedge A_1\}$ $\rightarrow Loc$
$\rightarrow Par$	$(P_2, A_2) = \{ok\colon P_1\}\{ad\colon A_2\}$	
$\rightarrow Dup$	float curp || uint32 num = α	
	$(P_2, A_2, Q_2) = \{ok\colon P_2\}\{ad\colon A_2\}\{ok\colon P_2\}$	
$\rightarrow Loc(\times 3)$	float curp || uint32 num = α || float curp2	
	$(P_3, A_3, Q_3) = \{ok\colon \exists c.curp = c \wedge P_2\}\{ad\colon \exists \alpha.num = \alpha \wedge A_2\}\{ok\colon \exists w.curp2 = w \wedge Q_2\}$	
$\rightarrow Loc, Wri, Loc$	uint32 ord || write(s1, num) || uint32 ord2	
	$(P_4, A_4, Q_4) = \{ok\colon \exists o.ord = o \wedge P_3\}$	
	$\{ad\colon \exists \alpha \exists s1_a^2.A_3(s1_a^2/s1_a) \wedge num = \alpha \wedge s1_a = (s1_a^2::\alpha)\}$	
	$\{ok\colon \exists o'.ord = o' \wedge Q_3\}$	
$\rightarrow Loc, Loc$	float dec || read(s1,guess1) || float dec2 {	
	$(P_5, A_5, Q_5) = \{ok\colon \exists d.dec = d \wedge P_4\}\{ad\colon A_4\}\{ok\colon \exists d^2.dec2 = d^2 \wedge Q_4\}$	
$\rightarrow Whi, Whi$	while (true) do || read(s1,guess1) || while (true) do	
	$(P_6, A_6, Q_6) = \{ok\colon true \wedge P_5\}\{ad\colon A_5\}\{ok\colon true \wedge Q_5\}$	
$\rightarrow Com$	read(s1, ord) || read(s1, guess1) { || read(s2, ord2)	
	$(P_7, A_7, Q_7) = \{ok\colon \exists \alpha \exists s1_p^2.P_6(s1_p^2/s1_p) \wedge s1_p = (s1_p^2::\alpha)\}$	
	$\{ad\colon \exists \alpha \exists s1_a^3.A_6(s1_a^3/s1_a) \wedge s1_a = (s1_a^3\backslash\alpha)\}\{ok\colon Q_6\}$	
$\rightarrow Read$	read(s1, ord) || read(s1, guess1) || read(s2, ord2)	
	$(P_8, A_8, Q_8) = \{ok\colon \exists \alpha \exists x \exists s1_p^3.P_7(x/ord, s1_p^3/s1_p) \wedge s1_p = (s1_p^3\backslash\alpha) \wedge ord = \alpha\}$	
	$\{ad\colon A_7\}\{ok\colon Q_7\}$	
$\rightarrow Assn$	dec = ord / V10MIL || read(s1, guess1) || read(s2, ord2)	
	$(P_9, A_9, Q_9) = \{ok\colon \exists d^2.P_8(d^2/dec) \wedge dec = ord/V10M\}\{ad\colon A_8\}\{ok\colon Q_8\}$	
$\rightarrow If$	if (ord <= 9MIL) curp = initp * (1 - dec) || read(s1, guess) || read(s2, ord2)	
	$(P_{10}, A_{10}, Q_{10}) = \{ok\colon \exists c^2.P_9[c^2/curp] \wedge ord \leq V9M \wedge curp = initp * (1 - dec)\}$	
	$\{ad\colon A_9\}\{ok\colon Q_9\}$	
$\rightarrow If$	else curp = initp / 10 || read(s1, guess1) || read(s2, ord2)	
	$(P_{11}, A_{11}, Q_{11}) = \{ok\colon \exists c^2.P_{10}[c^2/curp] \wedge ord > V9M \wedge curp = initp/10\}\{ad\colon A_9\}\{ok\colon Q_9\}$	
$\rightarrow Disj$	write(s1,curp) || read(s1, guess1) || read(s2, ord2)	
	$(P_{12}, A_{12}, Q_{12}) = \{ok\colon P_{11} \vee P_{10}\}\{ad\colon A_9\}\{ok\colon Q_9\}$	
$\rightarrow Wri$	write(s1,curp) || read(s1, guess1) || read(s2, ord2)	
	$(P_{13}, A_{13}, Q_{13}) = \{ok\colon \exists u \exists s1_p^4.P_{12}(s1_p^4/s1_p) \wedge curp = u \wedge s1_p = (s1_p^4::u)\}$	
	$\{ad\colon A_{12}\}\{ok\colon Q_{12}\}$	
$\rightarrow Com$	read(s1,ord) || read(s1, guess1) || read(s2, ord2)	
	$(P_{14}, A_{14}, Q_{14}) = \{ok\colon \exists u \exists s1_p^5.P_{13}(s1_p^5/s1_p) \wedge s1_p = (s1_p^5\backslash u)\}$	
	$\{ad\colon \exists u \exists s1_a^4.A_{13}(s1_a^4/s1_a) \wedge s1_a = (s1_a^4::u)\}\{ok\colon Q_{13}\}$	
$\rightarrow Read$	read(s1,ord) || read(s1, guess1) || read(s2, ord2)	
	$(P_{15}, A_{15}, Q_{15}) = \{ok\colon P_{14}\}\{ok\colon Q_{14}\}$	
	$\{ad\colon \exists g \exists u \exists s1_a^5.A_{14}(g/guess1, s1_a^5/s1_a) \wedge s1_a = (s1_a^5\backslash u) \wedge guess1 = u\}$	
$\rightarrow PBV$	read(s1,ord) || read(s1,guess1) || read(s2, ord2)	
	$(P_{16}, A_{16}, Q_{16}) = \{ok\colon \exists \alpha \exists f.\alpha > V9M \wedge initp = f \wedge (curp = initp/10)\}$	
	$\{ad\colon \exists u.guess1 = u\}\{ok\colon Q_{15}\}$	
$\rightarrow Adv.Cons.$	read(s1,ord) || read(s1,guess1) || read(s2, ord2)	
	$(P_{17}, A_{17}, Q_{17}) = \{ok\colon P_{16}\}\{ad\colon \exists \alpha \exists f \exists u.guess1 = u \wedge u = f/10 \wedge \alpha > V9M\}\{ok\colon Q_{16}\}$	
$\rightarrow Wri$	read(s1,ord) || write(s2,num) || read(s2, ord2)	
	$(P_{18}, A_{18}, Q_{18}) = \{ok\colon P_{17}\}\{ad\colon \exists \alpha \exists s2_a^2.A_{17}[s2_a^2/s2_a] \wedge num = \alpha \wedge s2_a = (s2_a^2::\alpha)\}$	
	$\{ok\colon Q_{17}\}$	
$\rightarrow Com$	read(s1,ord) || read(s2,guess2) || read(s2, ord2)	
	$(P_{19}, A_{19}, Q_{19}) = \{ok\colon P_{18}\}\{ad\colon \exists \alpha \exists s2_a^3.A_{18}[s2_a^3/s2_a] \wedge s2_a = (s2_a^3\backslash\alpha)\}$	
	$\{ok\colon \exists \alpha \exists s2_q^2.Q_{18}[s2_q^2/s2_q] \wedge s2_q = (s2_q^2::\alpha)\}$	
$\rightarrow Read$	read(s1,ord) || read(s2,guess2) || read(s2, ord2)	
	$(P_{20}, A_{20}, Q_{20}) = \{ok\colon P_{19}\}\{ad\colon A_{19}\}$	
	$\{ok\colon \exists \alpha \exists o2 \exists s2_q^3.Q_{19}[s2_q^3/s2_q, o2/ord2] \wedge s2_q = (s2_q^3\backslash\alpha) \wedge ord2 = \alpha\}$	
$\rightarrow Assn$	read(s1,ord) || read(s2,guess2) || dec2 = ord2 / V20MIL	
	$(P_{21}, A_{21}, Q_{21}) = \{ok\colon P_{20}\}\{ad\colon A_{20}\}\{ok\colon \exists d2.Q_{20}[d2/dec2] \wedge dec2 = ord2/V20M\}$	
$\rightarrow If$	read(s1,ord) || read(s2,guess2) || if (ord2 $\leq$ V18M) curp2 = initp * (1 - dec2)	
	$(P_{22}, A_{22}, Q_{22}) = \{ok\colon P_{21}\}\{ad\colon A_{21}\}$	
	$\{ok\colon \exists c2.Q_{21}[c2/curp2] \wedge (ord2 \leq V18M) \wedge curp2 = initp * (1 - dec2)\}$	
$\rightarrow If$	read(s1,ord) || read(s2,guess2) || else curp2 = initp / 10	
	$(P_{23}, A_{23}, Q_{23}) = \{ok\colon P_{22}\}\{ad\colon A_{22}\}$	
	$\{ok\colon \exists c2.Q_{21}[c2/curp2] \wedge (ord2 > V18M) \wedge (curp2 = initp/10)\}$	
$\rightarrow Disj$	read(s1,ord) || read(s2,guess2) || else curp2 = initp / 10	
	$(P_{24}, A_{24}, Q_{24}) = \{ok\colon P_{23}\}\{ad\colon A_{23}\}\{ok\colon Q_{22} \vee Q_{23}\}$	
$\rightarrow Wri$	read(s1,ord) || read(s2,guess2) || write(s2,curp2)	
	$(P_{25}, A_{25}, Q_{25}) = \{ok\colon P_{24}\}\{ad\colon A_{24}\}$	
	$\{ok\colon \exists c \exists s2_q^2.Q_{24}[s2_q^2/s2_q] \wedge (curp2 = c) \wedge s2_q = (s2_q^2::c)\}$	
$\rightarrow Com$	read(s1,ord) || read(s2,guess2) || read(s2, ord2)	
	$(P_{26}, A_{26}, Q_{26}) = \{ok\colon P_{25}\}\{ad\colon \exists c \exists s2_a^4.A_{25}[s2_a^4/s2_a] \wedge s2_a = (s2_a^4::c)\}$	
	$\{ok\colon \exists c \exists s2_q^3.Q_{25}[s2_q^3/s2_q] \wedge s2_q = (s2_q^3\backslash c)\}$	
$\rightarrow Read$	read(s1,ord) || read(s2, guess2) || read(s2, ord2)	
	$(P_{27}, A_{27}, Q_{27}) = \{ok\colon P_{26}\}\{ok\colon Q_{26}\}$	
	$\{ad\colon \exists g^2 \exists u^2 \exists s2_a^5.A_{26}(g^2/guess2, s2_a^5/s2_a) \wedge s2_a = (s2_a^5\backslash u^2) \wedge guess2 = u^2\}$	
$\rightarrow PBV$	read(s1,ord) || read(s2,guess2) || read(s2, ord2)	
	$(P_{28}, A_{28}, Q_{28}) = \{ok\colon P_{27}\}\{ad\colon \exists u^2.guess2 = u^2\}$	
	$\{ok\colon \exists \alpha \exists f.(\alpha > V18M) \wedge (initp = f) \wedge (curp2 = initp/10)\}$	
$\rightarrow Adv.Cons.$	read(s1,ord) || read(s2,guess2) || read(s2, ord2)	
	$(P_{29}, A_{29}, Q_{29}) = \{ok\colon P_{28}\}\{ad\colon \exists \alpha \exists f \exists u^2.(guess2 = u^2) \wedge (u = f/10) \wedge (\alpha > V18M)\}$	
	$\{ok\colon Q_{28}\}$	
$\rightarrow Succ$	read(s1,ord) || adv_assert(guess1 == guess2) || read(s2, ord2)	

quantities above which the price does not decrease anymore. In order to model this example in adversarial logic, we define and use a derived rule *Dup* at step (P_2, A_2) in Table 7. The *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 $c_1 = c_2$.

$$\textbf{Dup}\ \frac{[ok : P_1]c_1[ok : Q_1] \qquad [ok : P_1]c_1[ok : Q_1]}{[ok : P_1][ok : P_1]\ c_1\ ||\ c_1\ [ok : Q_1][ok : Q_1]}$$

The proof exhibits loop iterations at which the price converges, and symbolically compares return values in the adversary. Combining the parallel backward variant rule at (P_{16}, A_{16}, Q_{16}) followed by the adversarial consequence rule at (P_{17}, A_{17}, Q_{17}) gather *GetPrice* conditions, while this happens at (P_{28}, A_{28}, Q_{28}) and (P_{29}, A_{29}, Q_{29}) for function *GetPrice2*. This is possible without the adversary having preliminary knowledge of internal program values and state (such as the initial price), as long as the target program code is known.

5 Semantics

In this section, we develop a denotational semantics for adversarial logic. This semantics is defined compositionally for each of the rules of AL (Table 8).

Table 8. Relational denotational semantics for AL with transitions from state pairs (σ_p, σ_a) to (σ_q, σ_b) with $\epsilon, \epsilon_1, \epsilon_2 \in \{ok, ad\}$ and $\Sigma_x \in \{\Sigma_a, \Sigma_p\}$

$$
\begin{aligned}
[\![\mathbf{x = e}]\!]ok &= \{(\sigma, ((\sigma_p \mid x \mapsto [\![\mathrm{e}]\!]_{\sigma_p}), \sigma_a))\} \\
[\![\mathbf{x = e}]\!]ad &= \{(\sigma, (\sigma_p, (\sigma_a \mid x \mapsto [\![\mathrm{e}]\!]_{\sigma_a})))\} \\
[\![\mathbf{x = rand()}]\!]ok &= \{(\sigma, ((\sigma_p \mid x \mapsto v), \sigma_a))\} \\
[\![\mathbf{x = rand()}]\!]ad &= \{(\sigma, (\sigma_p, (\sigma_a \mid x \mapsto v)))\} \\
[\![\mathbf{adv\_assert\ B}]\!]ok &= \emptyset \\
[\![\mathbf{adv\_assert\ B}]\!]ad &= \{((\sigma_p, \sigma_a), \sigma) \mid [\![\mathrm{B}]\!]\sigma_a = \text{true}\ \} \uplus \{((\sigma_p, \sigma_a), \sigma) \mid [\![\mathrm{B}]\!]\sigma_a = \text{false}\} \\
[\![\mathbf{skip}]\!]\epsilon &= \{(\sigma, \sigma) \mid \sigma \in \Pi\} \\
[\![\mathbf{assume\ B}]\!]\epsilon &= \{(\sigma, \sigma) \mid [\![\mathrm{B}]\!]\sigma = \text{true}\} \\
[\![\mathbf{C}^*]\!]\epsilon &= \cup_{i \in \mathbb{N}} [\![C_i]\!]\epsilon \\
[\![\mathbf{C_1 + C_2}]\!]\epsilon &= [\![C_1]\!]\epsilon + [\![C_2]\!]\epsilon \\
[\![\mathbf{local\ x = e\ in\ C}]\!]ok &= \{(((\sigma_p \mid x \mapsto [\![\mathrm{e}]\!]_{\sigma_p}), \sigma_a), ((\sigma_q \mid x \mapsto v), \sigma_a)) \mid x \in Var, e \in Expr\} \\
[\![\mathbf{local\ x = e\ in\ C}]\!]ad &= \{((\sigma_p, (\sigma_a \mid x \mapsto [\![\mathrm{e}]\!]_{\sigma_a})), (\sigma_p, (\sigma_b \mid x \mapsto v))) \mid x \in Var, e \in Expr\} \\
[\![\mathbf{C_1; C_2}]\!]\epsilon &= \{(\sigma_1, \sigma_3) \mid (\sigma_1, \sigma_2) \in [\![C_1]\!]\epsilon \text{ and } (\sigma_2, \sigma_3) \in [\![C_2]\!]\epsilon\} \\
[\![\mathbf{C_1 || C_2}]\!](\epsilon_1, \epsilon_2) &= \{(\sigma_1, \sigma_1') \mid (\sigma_1, \sigma_1') \in [\![C_1]\!]\epsilon_1\} \cup \{(\sigma_2, \sigma_2') \mid (\sigma_2, \sigma_2') \in [\![C_2]\!]\epsilon_2\} \\
[\![\mathbf{read(s,x)}]\!]\epsilon &= \{((\sigma \mid s \mapsto (l::v)), (\sigma \mid x \mapsto v, s \mapsto l)) \mid s \in Chan, x \in Var\} \\
[\![\mathbf{write(s,x)}]\!]\epsilon &= \{((\sigma \mid s \mapsto l, x \mapsto v), (\sigma \mid s \mapsto (l::v))) \mid s \in Chan, x \in Var\} \\
[\![\mathbf{Com(C_1, C_2)}]\!](\epsilon_1, \epsilon_2) &= \{(((\sigma_1 \mid s \mapsto (v::l_1)), (\sigma_2 \mid s \mapsto l_2)), \\
&\quad ((\sigma_1 \mid s \mapsto l_1), (\sigma_2 \mid s \mapsto (l_2::v)))) \mid s \in Chan\}
\end{aligned}
$$

$$
\begin{gathered}
\Sigma_a : [Variables \to Values] \\
\Sigma_p : [Variables \to Values] \\
\Pi = \Sigma_p \times \Sigma_a \\
[\![\mathrm{B}]\!] : \Sigma_x \to Bool \\
\sigma = (\sigma_p, \sigma_a) : \Pi \\
[\![\mathrm{C}]\!] \subset \Pi \times \Pi
\end{gathered}
$$

We lay the groundwork to prove soundness of the logic and semantics by reminding some standard definitions.

Definition 1 (Post Image and Semantic Triples). *For any relation* $r \in \Sigma \times \Sigma$ *and predicate* $p \subseteq \Sigma$*:*

- *The post-image of* r, $post(r) \in P(\Sigma) \to P(\Sigma)$: $post(r)p = \{(\sigma' \mid \exists \sigma \in p.\ (\sigma, \sigma') \in r\}$
- *The over-approximate Hoare triple:* $\{p\}r\{q\}$ *iff* $post(r)p \subseteq q$
- *The under-approximate incorrectness triple:* $[p]r[q]$ *is true iff* $post(r)p \supseteq q$

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

Definition 2 (Adversarial Triples). *For any composed relation* $(ok, ad) \in \Pi \times \Pi$ *and predicates* $(p, a) \subseteq \Pi$ *with* p *the program predicate and* a *the adversarial predicate:*

- *The post-image of* (ok, ad) *noted* $post((ok, ad)) \in P(\Pi) \to P(\Pi)$*:*

$$post((ok, ad))(p, a) = \{(\sigma_q, \sigma_b) \mid \exists (\sigma_p, \sigma_a) \in (p, a).\ ((\sigma_p, \sigma_a), (\sigma_q, \sigma_b)) \in (ok, ad)\}$$

- *The under-approximate adversarial triple:*

$$[p][a](ok, ad)[q][b] \text{ is true iff } post((ok, ad))(p, a) \supseteq (q, b)$$

Conditions for membership $((\sigma_p, \sigma_a), (\sigma_q, \sigma_b)) \in (ok, ad)$ are defined as:

1. $(\sigma_p, \sigma_q) \in ok$ and $(\sigma_a, \sigma_b) \in ad$ if $VAR(\sigma_p) \cap VAR(\sigma_a) = \emptyset$
2. $(ok, ad)((\sigma_p, \sigma_a)) = (\sigma_q, \sigma_b)$ 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:*

- $\wedge\vee$ *symmetry:* $[\epsilon\colon p]c[\epsilon\colon q_1] \wedge [\epsilon\colon p]c[\epsilon\colon q_2] \iff [\epsilon\colon p]c[\epsilon\colon q_1 \vee q_2]$
- $\Uparrow\Downarrow$ *symmetry:* $[\epsilon\colon p \Rightarrow p'] \wedge [\epsilon\colon p]c[\epsilon\colon q] \wedge [\epsilon\colon q' \Rightarrow q] \iff [\epsilon\colon p']c[\epsilon\colon q']$

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 [1]. 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. $[pre\_ok\colon p][pre\_ad\colon a]\ C_1 \mid\mid C_2\ [post\_ok\colon q][post\_ad\colon b]$ *is true.*
2. *Every state in the conclusion is reachable from a state in the premises:* $\forall(\sigma_q, \sigma_b) \in (q, b)\ \exists\sigma_p \in p\ \exists\sigma_a \in a : ((\sigma_p, \sigma_a), (\sigma_q, \sigma_b)) \in (ok, ad)$

The characterization lemma in Adversarial Logic extends the one of Reverse Hoare Logic of de Vries and Koutavas [18] as inherited by Incorrectness Logic [1]. Sufficient conditions for the characterization lemma to hold can be decomposed into three subcases:

1. if $\sigma_a = \sigma_b : \forall(\sigma_q, \sigma_b) \in (q, b)\ \exists\sigma_p \in p : (\sigma_p, \sigma_q) \in ok$
2. if $\sigma_p = \sigma_q : \forall(\sigma_q, \sigma_b) \in (q, b)\ \exists\sigma_a \in a : (\sigma_a, \sigma_b) \in ad$
3. Otherwise: $\forall(\sigma_q, \sigma_b) \in (q, b)\ \exists(\sigma_p, \sigma_a) \in (p, a) : ((\sigma_p, \sigma_a), (\sigma_q, \sigma_b)) \in (ok, ad)$

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). $[ok\colon p][ad\colon a](C_1||C_2)[ok\colon q][ad\colon b]$ *is true iff the adversarial triple* $[p][a](\llbracket C_1||C_2 \rrbracket_{(ok,ad)})[q][b]$ *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 [18], which we hold true without proving it.

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

- $\sigma_p \in P(n/x) \iff (\sigma_p|x \to n) \in P$ *if* $x \in \sigma_p$
- $\sigma_a \in A(n/x) \iff (\sigma_a|x \to n) \in A$ *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 [18] by managing local variables using alpha-renaming, rather than using explicit substitution like O'Hearn [1]. This changes the soundness proof for the local variable rule and the assignment rule. For all symmetric cases involving ϵ, 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 (σ_p, σ_a) skip (σ_q, σ_b) and $[\epsilon\colon P]$ skip $[\epsilon\colon Q]$. Show that $\forall(\sigma_q, \sigma_b) \in [\epsilon\colon Q]\ \exists(\sigma_a, \sigma_p) \in [\epsilon\colon P]$. By skip rule, $P = Q$, so $[\epsilon\colon P]$ skip $[\epsilon\colon P]$ is true and $(\sigma_p, \sigma_a) = (\sigma_q, \sigma_b)$. Since $(\sigma_q, \sigma_b) \in [\epsilon\colon P]$ and $(\sigma_p, \sigma_a) = (\sigma_q, \sigma_b)$ then $(\sigma_p, \sigma_a) \in [\epsilon\colon P]$.

Proof (Constancy). Show that $\forall\sigma' \in [\epsilon\colon Q \wedge F]\ \exists\sigma \in [\epsilon\colon P \wedge F]$. By induction hypothesis, $\exists\sigma \in [\epsilon\colon P]$ such that $\sigma \to \sigma'$ and $[\epsilon\colon \sigma' \in Q]$. Since $Mod(c) \cap Free(F) = \emptyset$, $\sigma \to \sigma'$ preserves F. Therefore $\sigma \in [\epsilon\colon P \wedge F]$.

Proof (Assume). Let (σ_p, σ_a) assume B (σ_p, σ_a) and $[\epsilon\colon P]$ assume(B) $[\epsilon\colon P \wedge B]$. Show that $\forall(\sigma_q, \sigma_b) \in [\epsilon\colon P \wedge B]\ \exists(\sigma_p, \sigma_a) \in [\epsilon\colon P]$. Since $(\sigma_p, \sigma_a) = (\sigma_q, \sigma_b)$ by assume rule, $(\sigma_p, \sigma_a) \in P \wedge B$. By consequence rule, $(\sigma_p, \sigma_a) \in [\epsilon\colon P]$.

Proof (Rand). Assume (σ_p, σ_a) x = rand() (σ_q, σ_b) with $(\sigma_q, \sigma_b) = (\sigma_p \mid x \mapsto r, \sigma_a)$. Let $(\sigma_q, \sigma_b) \in [\epsilon\colon P(x/x') \wedge x = r]$ and show that $\forall(\sigma_q, \sigma_b)\ \exists(\sigma_p, \sigma_a) \in [\epsilon\colon P]$. Let us first cover the subcase where $x \in \sigma_p$ and $\epsilon = ok$ Take $(\sigma_p \mid x \mapsto n) \in [ok\colon P]$. By the substitution lemma, $\sigma_p \in [ok\colon P(n/x)]$. By assign rule, $\sigma_q \in [ok\colon P(r/x)]$. That is, $\sigma_p \in [ok\colon \exists x'.P(x'/x) \wedge x' = r]$. The second subcase where $x \in \sigma_a$ and $\epsilon = ad$ can be proved similarly.

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

Proof (Local). Let us first take the case where x $\in \sigma_q$. Show that $\forall(\sigma_q \mid x \mapsto v, \sigma_a) \in [ok\colon \exists x.Q]$ there is $(\sigma_p \mid x \mapsto [\![e]\!]_{\sigma_p}, \sigma_a) \in [ok\colon P]$. By the substitution lemma, $\sigma_q \in [ok\colon \exists x.Q(v/x)]$, that is $\sigma_q \in [ok\colon \exists x.Q]$ since x is bound. By induction hypothesis and executing backward, we obtain $\sigma_p \in [ok\colon P \wedge x = e]$. By the substitution lemma, we have $(\sigma_p \mid x \mapsto [\![e]\!]_{\sigma_p}) \in [ok\colon P(e/x)]$. Since $x \notin Free(P)$, we conclude $(\sigma_p \mid x \mapsto [\![e]\!]_{\sigma_p}) \in [ok\colon P]$. 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 $\sigma' \in [\epsilon\colon \exists x'\exists s'.P(s'/s, x'/x) \wedge (s = s'\backslash v) \wedge x = v]$ there is $(\sigma \mid s \mapsto (l{::}v)) \in [\epsilon\colon P]$. By the substitution lemma: $\sigma \in [\epsilon\colon \exists x'\exists s'.P(s'/s, x'/x) \wedge (s = s'\backslash v) \wedge x = v)(v/x)(l/s)]$ That is: $\sigma \in [\epsilon\colon \exists x'.\exists s'.P(s'/s, x'/x) \wedge (l = (s'\backslash v))]$. By rewriting s', we obtain $\sigma \in [\epsilon\colon \exists x.P((l{::}v)/s, x'/x)]$. Executing read backward, we get $(\sigma \mid s \mapsto (l{::}v), x \mapsto x') \in [\epsilon\colon P]$. We can conclude since $\{\sigma \mid s \mapsto (l{::}v), x \mapsto x'\} \subseteq \{\sigma \mid s \mapsto (l{::}v)\}$

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

Proof (Com). Assume $[\![Com(C_1, C_2)]\!]_{\epsilon_1,\epsilon_2} = \{(((\sigma_1 \mid s \mapsto (v::l_1)), (\sigma_2 \mid s \mapsto l_2)), (\sigma_1', \sigma_2'))\}$ 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 $(\sigma_1', \sigma_2') \in [\epsilon_1 : \exists s'.P(s'/s) \wedge s = (s'\backslash v)][\epsilon_2 : \exists s'.A(s'/s) \wedge s = (s'::v)]$ 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, $((\sigma_1 \mid s \mapsto (v::l_1)), (\sigma_2 \mid s \mapsto l_2)) \in [\epsilon_1 : P((v::l_1)/s)][\epsilon_2 : A(l_2/s)]$. Introducing s', we have $((\sigma_1 \mid s \mapsto (v::l_1)), (\sigma_2 \mid s \mapsto l_2)) \in [\epsilon_1 : \exists s'.P(s'/s) \wedge s' = (v::l_1)][\epsilon_2 : \exists s'.A(s'/s) \wedge s' = l_2]$. By $[\![Com(C_1, C_2)]\!]$ rule, $(\sigma_1', \sigma_2') \in [\epsilon_1 : \exists s'.P(s'/s) \wedge s' = (v::l_1) \wedge s = l_1]$ $[\epsilon_2 : \exists s'.A(s'/s) \wedge s' = l_2 \wedge s = (l_2::v)]$. Rewriting s using s' we now have: $(\sigma_1', \sigma_2') \in [\epsilon_1 : \exists s'.P(s'/s) \wedge s = (s'\backslash v)][\epsilon_2 : \exists s'.A(s'/s) \wedge s = (s'::v)]$.

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 [1] of AL.

Proof (Consequence). Immediate by logical definition and $\Uparrow\Downarrow$ symmetry [1] of AL.

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

Proof (Success). Assume (σ_p, σ_a) adv_assert(B) $\{(\sigma_q, \sigma_b) \mid [\![B]\!]_{\sigma_a} = true\}$ by $\uplus$ left subset. Success rule gives us that $[ad\colon P \wedge (P \Rightarrow B)]$ adv_assert(B) $[ad\colon P \wedge true]$. Show that $\forall (\sigma_q, \sigma_b) \in P \wedge true\ \exists (\sigma_p, \sigma_a) \in P \wedge (P \Rightarrow B)$. Success rule does not modify any variable of (σ_p, σ_a), therefore $(\sigma_p, \sigma_a) = (\sigma_q, \sigma_b)$ and $(\sigma_p, \sigma_a) \in P \wedge B$. Since $(P \wedge B) \iff P \wedge (P \Rightarrow B)$, we conclude that $(\sigma_p, \sigma_a) \in P \wedge (P \Rightarrow B)$.

Proof (Failure). Assume (σ_p, σ_a) adv_assert(B) $\{(\sigma_q, \sigma_b) \mid [\![B]\!]_{\sigma_a} = false\}$ by $\uplus$ right subset. Failure rule gives us that $[ad\colon P \wedge (P \Rightarrow \neg B)]$ adv_assert(B) $[ad\colon P \wedge \neg B]$. Show that $\forall (\sigma_q, \sigma_b) \in P \wedge \neg B\ \exists (\sigma_p, \sigma_a) \in P \wedge (P \Rightarrow \neg B)$. Failure rule does not modify any variable of (σ_p, σ_a), therefore $(\sigma_p, \sigma_a) = (\sigma_q, \sigma_b)$ and $(\sigma_p, \sigma_a) \in P \wedge \neg B$. Since $(P \wedge \neg B) \iff P \wedge (P \Rightarrow \neg B)$, we conclude that $(\sigma_p, \sigma_a) \in P \wedge (P \Rightarrow \neg B)$.

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

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

Proof (Case $n = m = 0$). Immediate by definition of **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 $[ok\colon P(n-1)][ad\colon A(m-1)]c_1||c_2[ok\colon P(n)][ad\colon A(m)]$ and there is a $(\sigma_{p(n-1)}, \sigma_{a(m-1)}) \in [ok\colon P(n-1)][ad\colon A(m-1)]$. We reuse the induction hypothesis several times going backward until we reach $(\sigma_{p0}, \sigma_{a0}) \in [ok\colon P(0)][ad\colon A(0)]$

Proof (Case $n \neq m$). By inductive hypothesis, it holds that $[ok\colon P(n)][ad\colon A(m)]c_1||c_2[ok\colon P(n+i)][ad\colon A(m+j)]$. Therefore, $\exists (\sigma_{q(n-i)}, \sigma_{b(m-j)}) \in [ok\colon P(n-i)][ad\colon A(m-j)]$. 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_n, j_m) where $i, j \in \{0, 1\}$. We have three subcases:

- $\delta(n,m) = (0,1)$ and $\exists (\sigma_{q(n)}, \sigma_{b(m-1)}) \in [ok\colon P(n)][ad\colon A(m-1)]$.
- $\delta(n,m) = (1,0)$ and $\exists (\sigma_{q(n-1)}, \sigma_{b(m)}) \in [ok\colon P(n-1)][ad\colon A(m)]$.
- $\delta(n,m) = (1,1)$ and $\exists (\sigma_{q(n-1)}, \sigma_{b(m-1)}) \in [ok\colon P(n-1)][ad\colon A(m-1)]$.

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:
  - $(\sigma_{q0}, \sigma_{b(m-j)}) \in [ok\colon P(0)][ad\colon A(m-j)]$.
  - For all remaining $(m-j)$ steps, we have $\delta(0,m) = (0,1)$
  - $(\sigma_{p0}, \sigma_{a0}) \xrightarrow{m-j} (\sigma_{p0}, \sigma_{b(m-j)}) \in [ok\colon P(0)][ad\colon A(m-j)]$
- The adversary reaches its initial condition before the program:
  - $(\sigma_{q(n-i)}, \sigma_{b0}) \in [ok\colon P(n-i)][ad\colon A(0)]$.
  - For all remaining $n-i$ steps, we have $\delta(n,0) = (1,0)$
  - $(\sigma_{p0}, \sigma_{a0}) \xrightarrow{n-i} (\sigma_{q(n-i)}, \sigma_{a0}) \in [ok\colon P(n-i)][ad\colon A(0)]$

6 Alternative Presentation

Different representations of program semantics can encode much of the same concepts as adversarial logic, albeit at different levels of abstractions. We briefly mention a couple of such representations without deep-diving into their respective theory.

6.1 Dynamic Logic

Many of the concepts put forward in this article can be expressed using the dynamic logic of Harel [16]. Let an adversarial system $S = (W, m, \pi)$ and its specification $\mathbb{AS} = [f_{s_0}, f_{s_1}, ..., f_{s_n}]$ with $\mathbb{F} \in \mathbb{AS}$ a list of formulae to be satisfied in order. A structure S can be defined as a triple (W, m, π) where W is a non-empty set of states, m is the state transition function, and π is a labeling function indicating in which state formulae in $\mathbb{F}$ hold.

$$S = (W, m, \pi) \widehat{=} \begin{cases} W = P \times A \\ m : W \rightarrow 2^{W \times W} \\ \pi : F \rightarrow 2^{W} \end{cases} \tag{1}$$

Satisfiability $S \vdash \mathbb{AS}$ can then be defined as conditions on the structure S.

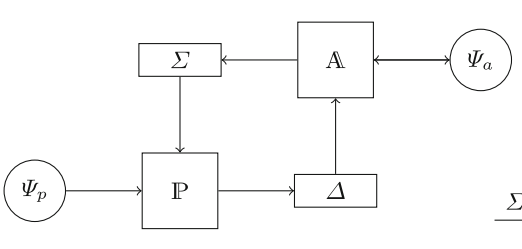
$$\exists z s_0, s_1, ..., s_n z \widehat{=} \begin{cases} s_0 = (\sigma_{p_0}, \sigma_{a_0}) \in \pi(f_{s_0}) \\ s_n = (\sigma_{p_n}, \sigma_{a_n}) \in \pi(f_{s_n}) \\ \forall j < n : ((\sigma_{p_j}, \sigma_{a_j}), (\sigma_{p_{j+1}}, \sigma_{a_{j+1}})) \in m(p, a) \\ \forall f_k \in \mathbb{AS} : \exists\, j_1 < j_2 < n : \\ \qquad -\sigma_{j_1} \notin \pi(f_k) \wedge \sigma_{j_1+1} \in \pi(f_k) \\ \qquad -\sigma_{j_2} \notin \pi(f_{k+1}) \wedge \sigma_{j_2+1} \in \pi(f_{k+1}) \end{cases} \tag{2}$$

The correspondence between dynamic logic [18] and incorrectness reasoning was remarked by O'Hearn [1]. This correspondence is preserved in adversarial logic with the change that every states is a couple (p, a) representing the product of the program state and the adversary state.

6.2 Information Systems

We now express adversarial logic concepts in the framework of domain theory [17]. In this formalism, we understand *adversarial systems* as a special kind of Scott's information system. We define an adversarial system $\mathbb{E} = \{\mathcal{D}, Con_{\mathcal{D}}, \vdash, \bot\}$ where $\mathcal{D} = \Psi_a \times \Sigma \times \Delta \times \Psi_p$ is the adversarial domain, $Con_{\mathcal{D}}$ is the set of all finite subsets of $\mathcal{D}$, $\bot$ is the least informative element of $\mathcal{D}$ and $\vdash$ is an entailment relation on $\mathcal{D}$. The entailment relation operates on a set of contexts Ψ_a, Ψ_p, Δ, and Σ, where (Fig. 1 and 2):

- Σ is the program input to execute the program with adversarial conditions.
- Ψ_p is the program context holding the symbolic program P.
- Δ is the program output produced by interpreting P with program input.
- Ψ_a is the adversarial context containing facts known by the adversary.



$$\frac{\Sigma \vdash A \qquad \Psi_p \vdash A \implies B}{\Delta \vdash B}\ \phi$$

$$\frac{\Delta \vdash B \qquad \Psi_p \vdash B \implies C}{\Psi_a \vdash C}\ \theta$$

$$\frac{\Sigma \vdash A \qquad \Psi_a, C \vdash A \times C \implies D}{\Sigma \vdash D}\ \delta$$

Fig. 1. Entailment relations for adversarial systems with $\mathbb{P}$ the program and $\mathbb{A}$ the adversary. Σ is the program input, Δ is the program output, Ψ_a is the adversarial context and Ψ_p is the program context.

$$\cfrac{\cfrac{\cfrac{\Sigma \vdash A \qquad \Psi_p \vdash A \implies B}{\Delta \vdash B}\ \phi \qquad \Psi_p \vdash B \implies C}{\Psi_a \vdash C}\ \theta \qquad \Sigma \vdash A \qquad \Psi_a \vdash A \times C \implies D}{\Sigma \vdash D}\ \delta$$

Fig. 2. Expected shape of proof tree in adversarial systems

We distinguish Ψ_a and Ψ_p to enforce that program knowledge is not shared to the adversary unless explicitly done so through the Ψ_a context. Entailment relation $\vdash$ is further partitioned into three sub-relations to distinguish each case of inference:

- $\vdash_\delta$: $\Sigma \times \Psi_a \to \Sigma$ is the adversarial entailment relation.
- $\vdash_\phi$: $\Sigma \times \Psi_p \to \Delta$ is the program entailment relation.
- $\vdash_\theta$: $\Delta \times \Psi_p \to \Psi_a$ is the knowledge entailment relation.

Adversarial entailment $\Sigma \times \Psi_a \vdash_\delta \Sigma$ derives next symbolic program input based on the previous input and the adversarial knowledge in context Ψ_a. Program entailment $\Sigma \times \Psi_p \vdash_\phi \Delta$ allows the program to compute an output value based on adversarial input (or from the program itself in case of recursive or internal procedures). Knowledge entailment $\Delta \times \Psi_p \vdash_\theta \Psi_a$ is the only rule which can increase adversarial knowledge Ψ_a. For example, adversarial knowledge of predicate $P(A, B)$ can be obtained based on an observable program output $C \in \Delta$ where $\vdash_\theta C \implies P(A, B)$ holds with $P(A, B) \in \Psi_p$, $A \in \Sigma$ and $B \in \Psi_p$. Reachability on $\mathbb{E}$ is defined as computing the least fixed point of the transitive closure of $\vdash$ to discover if the adversarial specification $\mathbb{A}\$$ is satisfiable. Formally, $\mathcal{D} \vdash \mathbb{A}\$ \iff \exists\ g \in \mathcal{D}$ such as $\{g\} \subseteq lfp_{\mathcal{D}}(\bot_{\mathcal{D}})$ and $g \vdash \mathbb{A}\$$. The oscillating bit protocol logic can be encoded in formula P as:

$$P = (s = 160) \wedge ((r^n = 0) \Rightarrow (v^n = s)) \wedge ((r^n = 1) \Rightarrow (v^n < s)) \\ \wedge ((r^n = 2) \Rightarrow (v^n > s))$$

The initial adversarial term can be encoded in formula A_0 as:

$$A_0 = (o = 0128) \wedge (s < v^n \Rightarrow (v^{n+1} = v^n - o^n \wedge o^{n+1} = o^n/2)) \\ \wedge (s > v^n \Rightarrow (v^{n+1} = v^n + o^n \wedge o^{n+1} = o^n/2))$$

Modeling the oscillating bit protocol in this framework is done in appendix.

7 Related Work

Related work in extended static checking and formal verification of software comes with a dense prior art, We enumerate a small fraction of the literature which directly influenced our thinking behind adversarial logic.

Incorrectness logic [1] is used as the starting point to formalize adversarial reasoning. In particular, AL borrows the backward variant rule of incorrectness logic and extend it to the parallel setting, a feature left out of scope of concurrent separation incorrectness logic by Raad et al. [14]. In the other hand, AL drops short-circuiting rules of IL, as program errors in AL must be carried transitively to determine the existence of attack paths. The characterization lemma used in under-approximate reasoning in IL and AL was introduced in reverse Hoare logic [18] and take its root in dynamic Logic [16].

Abstract Interpretation is a program analysis framework pioneered by Cousot and Cousot [12] and considered a reference technique in the verification of the absence of bugs. Abstract interpretation is practical [24] and comes with a rich legacy of applications including the creation of abstractions for theorem proving [25], model checking [26], worst-case execution time analysis [27], thread-modular analysis for concurrent programs [28], and input data tracking [29]. In comparison, adversarial logic (and incorrectness logic before it) cannot guarantee the absence of bugs due to its fundamentally under-approximate nature focused on eliminating false positives at the expense of false negatives. Incorrectness principles have been captured in the abstract interpretation framework by the local completeness logic LCL [30], and algebras of correctness and incorrectness can provide a unified formalism to connect both approaches [31].

Process calculus [23] is a well-established formalism to reason about parallel communicating programs and program equivalence using bisimulation. Abadi and Blanchet [32] designed the spi-calculus to verify secrecy properties of cryptographic protocols in the symbolic model. To the same goal, the proverif [33] tool by Blanchet et al. implements the Dolev-Yao model [2] with explicit attacker. It may be possible to extend proverif to include arithmetic in its specifications language, which is required to implement the examples of this paper.

Separation logic is a well-established logic to encode heap reasoning in program analysis. Separation logic comes in both over-approximate [21] and under-approximate [13] flavors. Combined with parallel constructs, separation logic leads to concurrent separation logics [34] and concurrent incorrectness separation logic [14]. Adversarial Logic provides a limited kind of separation between variables of parallel processes without requiring an explicit separating conjunction. Encoding separation expressiveness without the star operator is not unseen, and was previously implemented in the framework of linear maps [35]. Adding support for heap reasoning is a natural next step for adversarial logic.

Automated bug finding by symbolic execution [36,37], white-box fuzz testing [38], and extended static checkers [39] using SMT solvers [40] are often used to maximize code coverage in static and dynamic program analysis. These tools typically focus on checking sequential properties of non-interactive parser-like code [41], leaving concurrency out of scope. Symbolic execution using SMT

solvers have known scalability issues with path explosions in loops and constraint tracking in deep paths. Adversarial logic addresses these issues by only requiring a subset of paths to be analyzed sufficient to prove the presence of exploitable bugs. AL implements a flavor of concurrent symbolic execution where symbolic variables are introduced by the adversary to drive attack search without requiring knowledge of internal program state. As such, AL can express adversarial symbolic execution [42] as used to detect concurrency-related cache timing leaks.

Automated exploit generation (AEG [43]) leverages preconditioned symbolic execution to craft a sufficient program condition to exploit stack-based buffer overflow security vulnerabilities. Specific domains of heap vulnerabilities for interpreted languages have been demonstrated practical to attack by Heelan et al. [44]. Concepts of adversarial logic could possibly be added to extend AEG, such as for tackling information disclosure vulnerabilities as illustrated by the Oscillating Bit Protocol example in Sect. 2.

8 Conclusion and Future Work

Adversarial logic (AL) is a new under-approximate logic extending incorrectness logic [1] to perform exploitability analysis of software bugs. Reasoning about accumulated error in programs is critical to understand the severity of security issues and prioritize bug fixing accordingly. This new logic can be used to discover attacks which require a deeper level of interaction with the program, such as subtle information disclosure attacks in interactive protocol loops. We provided a denotational semantics and proved the soundness of adversarial logic showing that all exhibited attack traces in AL are true positives. In the future, embedding adversarial logic principles in concurrent incorrectness separation logic [14] will extend adversarial logic with heap reasoning, so AL can also be used to perform exploitability analysis of pointer bugs.

Acknowledgments. The author thanks Peter O'Hearn, Azalea Raad and Samantha Gottlieb for their useful reviews of this paper.

References

1. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL), 1–32 (2019)
2. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Trans. Inf. Theory **29**(2), 198–208 (1983)
3. Vyukov, D.: Syzkaller (2015)
4. Serebryany, K.:Continuous fuzzing with libfuzzer and addresssanitizer. In: 2016 IEEE Cybersecurity Development (SecDev), pp. 157–157. IEEE (2016)
5. Project, T.A.: Apache log4j security vulnerabilities (2022)
6. Durumeric, Z., et al.: The matter of heartbleed, pp. 475–488 (2014)
7. Bratus, S., Locasto, M.E., Patterson, M.L., Sassaman, L., Shubina, A.: Exploit programming: from buffer overflows to weird machines and theory of computation. USENIX; Login **36**(6), 13–21 (2011)

8. Dowd, M.: Sendmail release notes for the crackaddr vulnerability (2003)
9. Sotirov, A.: Apache OpenSSL heap overflow exploit (2002)
10. Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., Mangard, S.: KASLR is dead: long live KASLR. In: Bodden, E., Payer, M., Athanasopoulos, E. (eds.) ESSoS 2017. LNCS, vol. 10379, pp. 161–176. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62105-0_11
11. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252 (1977)
13. Raad, A., Berdine, J., Dang, H.-H., Dreyer, D., O'Hearn, P., Villard, J.: Local reasoning about the presence of bugs: incorrectness separation logic. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 225–252. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_14
14. Raad, A., Berdine, J., Dreyer, D., O'Hearn, P.W.: Concurrent incorrectness separation logic (2022)
15. Blazytko, T., et al.: {AURORA}: Statistical crash analysis for automated root cause explanation. In: 29th {USENIX} Security Symposium ({USENIX} Security 2020), pp. 235–252 (2020)
16. Harel, D., et al.: First-order dynamic logic (1979)
17. Scott, D.S.: Domains for denotational semantics. In: Nielsen, M., Schmidt, E.M. (eds.) ICALP 1982. LNCS, vol. 140, pp. 577–610. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0012801
18. de Vries, E., Koutavas, V.: Reverse Hoare logic. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 155–171. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24690-6_12
19. Kocher, P., et al.: Spectre attacks: exploiting speculative execution. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 1–19. IEEE (2019)
20. Lipp, M., et al.: Meltdown. arXiv preprint arXiv:1801.01207 (2018)
21. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, pp. 55–74. IEEE (2002)
22. Cardwell, J.R.: Ipv6 security issues in Linux and FreeBSD kernels: a 20-year retrospective (2018)
23. Milner, R.: Communicating and Mobile Systems: The PI Calculus. Cambridge University Press, Cambridge (1999)
24. Blanchet, B., et al.: A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pp. 196–207 (2003)
25. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. J. Logic Program. **13**(2–3), 103–179 (1992)
26. Cousot, P., Cousot, R.: Refining model checking by abstract interpretation. Autom. Softw. Eng. **6**(1), 69–95 (1999)
27. Wilhelm, R., et al.: The worst-case execution-time problem-overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. (TECS) **7**(3), 1–53 (2008)
28. Miné, A.: Relational thread-modular static value analysis by abstract interpretation. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 39–58. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54013-4_3

29. Urban, C., Müller, P.: An abstract interpretation framework for input data usage. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 683–710. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_24
30. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: A logic for locally complete abstract interpretations. In: 2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), pp. 1–13. IEEE (2021)
31. Möller, B., O'Hearn, P., Hoare, T.: On algebra of program correctness and incorrectness. In: Fahrenberg, U., Gehrke, M., Santocanale, L., Winter, M. (eds.) RAMiCS 2021. LNCS, vol. 13027, pp. 325–343. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88701-8_20
32. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: the SPI calculus. Inf. Comput. **148**(1), 1–70 (1999)
33. Blanchet, B., Smyth, B., Cheval, V., Sylvestre, M.: Proverif 2.00: automatic cryptographic protocol verifier, user manual and tutorial, pp. 05–16 (2018)
34. Brookes, S., O'Hearn, P.W.: Concurrent separation logic. ACM SIGLOG News **3**(3), 47–65 (2016)
35. Lahiri, S.K., Qadeer, S., Walker, D.: Linear maps. In: Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification, pp. 3–14 (2011)
36. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: automatically generating inputs of death. ACM Trans. Inf. Syst. Secur. (TISSEC) **12**(2), 1–38 (2008)
37. Cadar, C., Dunbar, D., Engler, D.R., et al.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, vol. 8, pp. 209–224 (2008)
38. Godefroid, P., Levin, M.Y., Molnar, D.: Sage: whitebox fuzzing for security testing. Queue **10**(1), 20:20–20:27 (2012)
39. Ball, T., Hackett, B., Lahiri, S.K., Qadeer, S., Vanegue, J.: Towards scalable modular checking of user-defined properties. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 1–24. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15057-9_1
40. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
41. Vanegue, J., Lahiri, S.K.: Towards practical reactive security audit using extended static checkers. In: 2013 IEEE Symposium on Security and Privacy, pp. 33–47. IEEE (2013)
42. Guo, S., Wu, M., Wang, C.: Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 377–388 (2018)
43. Brumley, D., Cha, S.K., Avgerinos, T.: Automated exploit generation. US Patent App. 13/481,248 (2012)
44. Heelan, S., Melham, T., Kroening, D.: Automatic heap layout manipulation for exploitation, pp. 763–779 (2018)