A General Approach to Under-Approximate Reasoning About Concurrent Programs

Azalea Raad
Imperial College London, UK

Julien Vanegue
Bloomberg, New York, NY, USA

Josh Berdine
Skiplabs, London, UK

Peter O'Hearn
University College London, UK
Lacework, London, UK

Abstract

There is a large body of work on concurrent reasoning including Rely-Guarantee (RG) and Concurrent Separation Logics. These theories are *over-approximate*: a proof identifies a *superset* of program behaviours and thus implies the absence of certain bugs. However, failure to find a proof does not imply their presence (leading to *false positives* in over-approximate tools). We describe a general theory of *under-approximate* reasoning for concurrency. Our theory incorporates ideas from Concurrent Incorrectness Separation Logic and RG based on a subset rather than a superset of interleavings. A strong motivation of our work is detecting *software exploits*; we do this by developing *concurrent adversarial separation logic* (CASL), and use CASL to detect *information disclosure attacks* that uncover sensitive data (e.g. passwords) and *out-of-bounds attacks* that corrupt data. We also illustrate our approach with classic concurrency idioms that go beyond prior under-approximate theories which we believe can inform the design of future concurrent bug detection tools.

2012 ACM Subject Classification Theory of computation → Separation logic; Theory of computation → Programming logic; Theory of computation → Program analysis; Security and privacy → Logic and verification

Keywords and phrases Under-approximate reasoning, incorrectness logic, bug detection, software exploits, separation logic

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.25

Related Version *Extended Version*: `https://www.soundandcomplete.org/papers/CONCUR2023/CASL/appendix.pdf`

Funding *Azalea Raad*: UKRI Future Leaders Fellowship under grant number MR/V024299/1

1 Introduction

Incorrectness Logic (IL) [16] presents a formal foundation for proving the *presence* of bugs using *under-approximation*, i.e. focusing on a *subset* of behaviours to ensure one detects only *true positives* (real bugs) rather than *false positives* (spurious bug reports). This is in contrast to verification frameworks proving the *absence* of bugs using *over-approximation*, where a *superset* of behaviours is considered. The key advantage of under-approximation is that tools underpinned by it are accompanied by a *no-false-positives* (NFP) theorem *for free*, ensuring all bugs reported are real bugs. This has culminated in a successful trend in automated static analysis tools that use under-approximation for bug detection, e.g. RacerD [3] for data race detection in Java programs, the work of Brotherston et al. [4] for deadlock detection, and Pulse-X [13] which uses the under-approximate theory of ISL (incorrectness separation logic, an IL extension) [17] for detecting memory safety bugs such as use-after-free errors. All three tools are currently industrially deployed and are state-of-the art techniques: RacerD significantly outperforms other race detectors in terms of bugs found and fixed, while Pulse-X has a higher fix-rate than the industrial Infer tool [7] used widely at Meta, Amazon and Microsoft. IL and ISL, though, only support bug detection in *sequential* programs.

We present *concurrent adversarial separation logic* (CASL, pronounced "castle"), a general, under-approximate framework for detecting concurrency bugs and exploits, including a hitherto unsupported class of bugs. Inspired by adversarial logic [22], we model a vulnerable program $\mathsf{C_v}$ and its attacker (adversarial) $\mathsf{C_a}$ as the concurrent program $\mathsf{C_a} \,||\, \mathsf{C_v}$, and use the compositional principles of CASL to detect vulnerabilities in $\mathsf{C_v}$. CASL is a *parametric* framework that can be instantiated for a range of bugs/exploits. CASL combines under-approximation with ideas from RGSep [20] and concurrent separation logic (CSL) [15] – we chose RGSep rather than rely-guarantee [11] for compositionality (see p. 7). However, CASL does not merely replace over- with under-approximation in RGSep/CSL: CASL includes an additional component witnessing (*under-approximating*) the interleavings leading to bugs.

CASL builds on *concurrent incorrectness separation logic* (CISL) [18]. However, while CISL was designed to capture the reasoning in cutting-edge tools such as RacerD, CASL explicitly goes beyond these tools. Put differently, CISL aspired to be a *specialised* theory of concurrent under-approximation, oriented to existing tools (and inheriting their limitations), whereas CASL aspires to be more *general.* In particular, in our private communication with CISL authors they have confirmed two key limitations of CISL. First, CISL can detect certain bugs compositionally only by encoding buggy executions as normal ones. While this is sufficient for bugs where encountering a bug does not force the program to terminate (e.g. data races), it cannot handle bugs with *short-circuiting semantics*, e.g. null pointer exceptions, where the execution is halted on encountering the bug (see §2 for details). Second and more significantly, CISL cannot *compositionally* detect a large class of bugs, *data-dependent* bugs, where a bug occurs only under certain interleavings and concurrent threads affect the control flow of one another. To see this, consider the program $\mathsf{P} \triangleq x := 1 \,||\, a := x; \mathsf{if}\ (a)\ \mathsf{error}$, where the left thread, τ_1, writes 1 to x, the right thread, τ_2, reads the value of x in a and subsequently errors if $a \neq 0$. That is, the error occurs only in interleavings where τ_1 is executed before τ_2, and the two threads synchronise on the value of x; i.e. τ_1 affects the control flow of τ_2 and the error occurrence is *dependent* on the *data* exchange between the threads.

Such data-dependency is rather prevalent as threads often synchronise via *data exchange.* Moreover, a large number of security-breaking *software exploits* are data-dependent bugs. An exploit (or *attack*) is code that takes advantage of a bug in a vulnerable program to cause unintended or erroneous behaviours. *Vulnerabilities* are bugs that lead to critical security compromises (e.g. leaking secrets or elevating privileges). Distinguishing vulnerabilities from benign bugs is a growing problem; understanding the exploitability of bugs is a time-consuming process requiring expert involvement, and large software vendors rely on automated exploitability analysis to prioritise vulnerability fixing among a sheer number of bugs. Rectifying vulnerabilities in the field requires expensive software mitigations (e.g. addressing Meltdown [14]) and/or large-scale recalls. It is thus increasingly important to detect vulnerabilities pre-emptively during development to avoid costly patches and breaches.

To our knowledge, CASL is the *first* under-approximate theory that can detect *all* categories of concurrency bugs (including data-dependent ones) *compositionally* (by reasoning about each thread in isolation). CASL is strictly stronger than CISL and supports all CISL reasoning principles. Moreover, CASL is the *first* under-approximate and compositional theory for exploit detection. We instantiate CASL to detect *information disclosure attacks* that uncover sensitive data (e.g. Heartbleed [8]) and *out-of-bounds attacks* that corrupt data (e.g. zero allocation [21]). Thanks to CASL soundness, each CASL instance is automatically accompanied by an NFP theorem: all bugs/exploits identified by it are true positives.

Contributions and Outline. Our contributions (detailed in §2) are as follows. We present CASL (§3) and prove it sound, with the full proof given in the accompanying technical appendix [19]. We instantiate CASL to detect information disclosure attacks on stacks (§4) and heaps [19, §C] and memory safety attacks [19, §D]. We also develop an under-approximate analogue of RG that is simpler but less expressive than CASL [19, §E and §F]. We discuss related work in §5.

2 Overview

CISL and Its Limitations. CISL [18] is an under-approximate logic for detecting bugs in concurrent programs with a built-in *no-false-positives theorem* ensuring all bugs detected are true bugs. Specifically, CISL allows one to prove triples of the form $[p]\ \mathsf{C}\ [\epsilon : q]$, stating that *every* state in q is reachable by executing C starting in *some* state in p, under the (exit) condition ϵ that may be either ok for normal (non-erroneous) executions, or $\epsilon \in \textsc{ErExit}$ for erroneous executions, where ErExit contains erroneous conditions. The CISL authors identify *global* bugs as those that are due to the interaction between two or more concurrent threads and arise only under certain interleavings. To see this, consider the examples below [18], where we write τ_1 and τ_2 for the left and right threads in each example, respectively:

$$\textsc{l}\colon \mathsf{free}(x) \;\Big|\Big|\; \textsc{l}'\colon \mathsf{free}(x) \qquad (\textsc{DataAgn}) \qquad\qquad \begin{array}{l}\mathsf{free}(x);\\ {[z]} := 1;\end{array} \Bigg|\Bigg| \begin{array}{l} a := 0;\ a := [z];\\ \mathsf{if}\ (a{=}1)\ \textsc{l}\colon [x] := 1\end{array} \qquad (\textsc{DataDep})$$

In an interleaving of DataAgn in which τ_1 is executed after (resp. before) τ_2, a double-free bug is reached at L (resp. L$'$). Analogously, in a DataDep interleaving where τ_2 is executed after τ_1, value 1 is read from z in a, the condition of if is met and thus we reach a use-after-free bug at L. Raad et al. [18] categorise global bugs as either *data-agnostic* or *data-dependent*, denoting whether concurrent threads contributing to a global bug may affect the *control flow* of one another. For instance, the bug at L in DataDep is data-dependent as τ_1 may affect the control flow of τ_2: the value read in $a := [z]$, and subsequently the condition of if and whether $\textsc{l}\colon [x] := 1$ is executed depend on whether τ_2 executes $a := [z]$ before or after τ_1 executes $[z] := 1$. By contrast, the threads in DataAgn cannot affect the control flow of one another; hence the bugs at L and L$'$ are data-agnostic.

$$\textsc{CISL-Par}\ \frac{[P_1]\,\mathsf{C}_1\,[ok\!:\!Q_1] \qquad [P_2]\,\mathsf{C}_2\,[ok\!:\!Q_2]}{[P_1 * P_2]\ \mathsf{C}_1 \,||\, \mathsf{C}_2\ [ok\colon Q_1 * Q_2]}$$

In *certain cases*, CISL can detect data-agnostic bugs compositionally (i.e. by analysing each thread in isolation) by encoding buggy executions as normal (ok) ones and then using the CISL-Par rule shown across. In particular, when the targeted bugs do not manifest *short-circuiting* (where bug encounter halts execution, e.g. a null-pointer exception), then buggy executions can be encoded as normal ones and subsequently detected compositionally using CISL-Par. For instance, when a data-agnostic data race is encountered, execution is not halted (though program behaviour may be undefined), and thus data races can be encoded as normal executions and detected by CISL-Par. By contrast, in the case of data-agnostic errors such as null-pointer exceptions, the execution is halted (i.e. short-circuited) and thus can no longer be encoded as normal executions that terminate. As such, *CISL cannot detect data-agnostic bugs with short-circuiting semantics compositionally.*

$$dom(\mathcal{G}_1) = \{\alpha_1, \alpha_2\} \qquad dom(\mathcal{G}_2) = \{\alpha'_1, \alpha'_2\} \qquad \mathcal{R}_1 \triangleq \mathcal{G}_2 \qquad \mathcal{R}_2 \triangleq \mathcal{G}_1 \qquad \theta \triangleq [\alpha_1, \alpha_2, \alpha'_1, \alpha'_2]$$

$$\mathcal{G}_1(\alpha_1) \triangleq (x \Mapsto l_x * l_x \mapsto v_x,\ ok,\ x \Mapsto l_x * l_x \not\mapsto) \qquad \mathcal{G}_2(\alpha'_1) \triangleq (z \Mapsto l_z * l_z \mapsto 1,\ ok,\ z \Mapsto l_z * l_z \mapsto 1)$$

$$\mathcal{G}_1(\alpha_2) \triangleq (z \Mapsto l_z * l_z \mapsto v_z,\ ok,\ z \Mapsto l_z * l_z \mapsto 1) \qquad \mathcal{G}_2(\alpha'_2) \triangleq (x \Mapsto l_x * l_x \not\mapsto,\ er,\ x \Mapsto l_x * l_x \not\mapsto)$$

$$\emptyset, \mathcal{G}_1 \cup \mathcal{G}_2, \{[\,]\} \vdash \left[a \Mapsto v_a * \boxed{x \Mapsto l_x * l_x \mapsto v_x * z \Mapsto l_z * l_z \mapsto v_z}\right] \quad //\ \textsc{Par}$$

$\mathcal{R}_1, \mathcal{G}_1,$

$\{[\,]\} \vdash \left[\boxed{x \Mapsto l_x * l_x \mapsto v_x * z \Mapsto l_z * l_z \mapsto v_z}\right]$

1. $\mathsf{free}(x);$ $//\ \textsc{Atom}, \text{MS-}\textsc{Free}$

$\{[\alpha_1]\} \vdash \left[ok\colon \boxed{\begin{array}{l} x \Mapsto l_x * l_x \not\mapsto \\ * z \Mapsto l_z * l_z \mapsto v_z \end{array}}\right]$

2. $[z] := 1;$ $//\ \textsc{Atom}, \text{MS-}\textsc{Write}$

$\{[\alpha_1, \alpha_2]\} \vdash \left[ok\colon \boxed{\begin{array}{l} x \Mapsto l_x * l_x \not\mapsto \\ * z \Mapsto l_z * l_z \mapsto 1 \end{array}}\right]$

3. $//\ \textsc{EnvR}$

$\{[\alpha_1, \alpha_2, \alpha'_1]\} \vdash \left[ok\colon \boxed{\begin{array}{l} x \Mapsto l_x * l_x \not\mapsto \\ * z \Mapsto l_z * l_z \mapsto 1 \end{array}}\right]$

4. $//\ \textsc{EnvR}$

$\{\theta\} \vdash \left[er\colon \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto 1}\right]$

$\|$

$\mathcal{R}_2, \mathcal{G}_2,$

$\{[\,]\} \vdash \left[a \Mapsto v_a * \boxed{x \Mapsto l_x * l_x \mapsto v_x * z \Mapsto l_z * l_z \mapsto v_z}\right]$

5. $//\ \textsc{EnvL}$

$\{[\alpha_1]\} \vdash \left[ok\colon a \Mapsto v_a * \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto v_z}\right]$

6. $//\ \textsc{EnvL}$

$\{[\alpha_1, \alpha_2]\} \vdash \left[ok\colon a \Mapsto v_a * \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto 1}\right]$

7. $a := 0;$ $//\ \textsc{AtomLocal}, \text{MS-}\textsc{AssignVal}$

$\{[\alpha_1, \alpha_2]\} \vdash \left[ok\colon a \Mapsto 0 * \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto 1}\right]$

8. $a := [z];$ $//\ \textsc{Atom}, \text{MS-}\textsc{Read}$

$\{[\alpha_1, \alpha_2, \alpha'_1]\} \vdash \left[ok\colon a \Mapsto 1 * \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto 1}\right]$

9. $\mathsf{if}\ (a = 1)\ [x] := 1$ $//\ \textsc{Atom}, \text{MS-}\textsc{WriteUAF}$

$\{\theta\} \vdash \left[er\colon a \Mapsto 1 * \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto 1}\right]$

$$\emptyset, \mathcal{G}_1 \cup \mathcal{G}_2, \{\theta\} \vdash \left[er\colon a \Mapsto 1 * \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto 1}\right]$$

Figure 1 CASL proof of DataDep; the // denote CASL rules applied at each step. The $\mathcal{R}_1, \mathcal{G}_1$ and $\mathcal{R}_2, \mathcal{G}_2$ are not repeated at each step as they are unchanged.

More significantly, however, CISL is altogether *unable to detect data-dependent bugs compositionally.* Consider the data-dependent use-after-free bug at L in DataDep. As discussed, this bug occurs when τ_2 is executed after τ_1 is fully executed (i.e. 1 is written to z and x is deallocated). That is, for τ_2 to read 1 for z it must somehow infer that τ_1 writes 1 to z; this is not possible without having knowledge of the environment. This is reminiscent of *rely-guarantee* (RG) reasoning [11], where the environment behaviour is abstracted as a relation describing how it may manipulate the state. As RG only supports global and not compositional reasoning about states, RGSep [20] was developed by combining RG with separation logic to support state compositionality. We thus develop CASL as an under-approximate analogue of RGSep for bug catching (see p. 7 for a discussion on RGSep/RG).

2.1 CASL for Compositional Bug Detection

In CASL we prove under-approximate triples of the form $\mathcal{R}, \mathcal{G}, \Theta \vdash [P]\ \mathsf{C}\ [\epsilon : Q]$, stating that every post-*world* $w_q \in Q$ is reached by running C on some pre-world $w_p \in P$, with $\mathcal{R}$, $\mathcal{G}$ and Θ described shortly. Each CASL world w is a pair (l, g), where $l \in \textsc{State}$ is the *local* state not accessible by the environment, while $g \in \textsc{State}$ is the *shared* (global) state accessible by all threads. We define CASL in a general, parametric way that can be instantiated for different use cases. As such, the choice of the underlying states, State, is a parameter to be instantiated accordingly. For instance, in what follows we instantiate CASL to detect the use-after-free bug in DataDep, where we define states as $\textsc{State} \triangleq \textsc{Stack} \times \textsc{Heap}$ (see §3), i.e. each state comprises a variable store and a heap.

For better readability, we use P, Q, R as meta-variable for sets of worlds and p, q, r for sets of states. We write $p * \boxed{q}$ for sets of worlds (l, g) where the local state is given by p $(l \in p)$ and the shared state is given by q $(g \in q)$. Given P and Q describing e.g. the worlds of two different threads, the composition $P * Q$ is defined component-wise on the local and shared states. More concretely, as local states are thread-private, they are combined via the composition operator $*$ on states in STATE (also supplied as a CASL parameter). On the other hand, as shared states are globally visible to all threads, the views of different threads of the shared state must agree and thus shared states are combined via conjunction $(\wedge)$. That is, given $P \triangleq p * \boxed{p'}$ and $Q \triangleq q * \boxed{q'}$, then $P * Q \triangleq p * q * \boxed{p' \wedge q'}$.

The *rely* relation, $\mathcal{R}$, describes how the environment threads may access/update the shared state, while the *guarantee* relation, $\mathcal{G}$, describes how the threads in C may do so. Specifically, both $\mathcal{R}$ and $\mathcal{G}$ are maps of *actions*: given $\mathcal{G}(\alpha) \triangleq (p, \epsilon, q)$, the α denotes an *action identifier* and (p, ϵ, q) denotes its effect, where p, q are sets of shared states and ϵ is an exit condition. Lastly, Θ denotes a set of *traces* (interleavings), such that each trace $\theta \in \Theta$ is a sequence of actions taken by the threads in C or the environment, i.e. the actions in $dom(\mathcal{G})$ and $dom(\mathcal{R})$. In particular, $\mathcal{R}, \mathcal{G}, \Theta \vdash [P]\ \mathsf{C}\ [\epsilon : Q]$ states that for all traces $\theta \in \Theta$, each world in Q is reachable by executing C on some world in P culminating in θ, where the effects of the threads in C (resp. in the environment of C) on the shared state are given by $\mathcal{G}$ and $\mathcal{R}$, respectively. We shortly elaborate on this through an example.

CASL for Detecting Data-Dependent Bugs. Although CASL can detect all bugs identified by Raad et al. [18], we focus on using CASL for data-dependent bugs as they cannot be handled by the state-of-the-art CISL framework. In Fig. 1 we present a CASL proof sketch of the bug in DATADEP. Let us write τ_1 and τ_2 for the left and right threads in Fig. 1, respectively. Variables x and z are accessed by both threads and are thus *shared*, whereas a is accessed by τ_2 only and is *local*. Similarly, heap locations l_x and l_z (recorded in x and z) are shared as they are accessed by both threads. This is denoted by $P_2 \triangleq a \Mapsto v_a * \boxed{x \Mapsto l_x * l_x \mapsto v_x * z \Mapsto l_z * l_z \mapsto v_z}$ in the pre-condition of τ_2 in Fig. 1, describing worlds in which the local state is $a \Mapsto v_a$ (stating that stack variable a records value v_a), and the global state is $x \Mapsto l_x * l_x \mapsto v_x * z \Mapsto l_z * l_z \mapsto v_z$ – note that we use the $\Mapsto$ and $\mapsto$ arrows for stack and heap resources, respectively. By contrast, the τ_1 precondition is $P_1 \triangleq \boxed{x \Mapsto l_x * l_x \mapsto v_x * z \Mapsto l_z * l_z \mapsto v_z}$, comprising only shared resources and no local resources.

The actions in $\mathcal{G}_1$ (resp. $\mathcal{G}_2$), defined at the top of Fig. 1, describe the effect of τ_1 (resp. τ_2) on the shared state. For instance, $\mathcal{G}_1(\alpha_1)$ describes executing $\mathsf{free}(x)$ by τ_1: when the shared state contains $x \Mapsto l_x * l_x \mapsto v_x$, i.e. a *sub-part* of the shared state satisfies $x \Mapsto l_x * l_x \mapsto v_x$, then $\mathsf{free}(x)$ terminates normally (ok) and deallocates x, updating this sub-part to $x \Mapsto l_x * l_x \not\mapsto$, denoting that l_x is deallocated. Dually, the actions in $\mathcal{R}_1$ (resp. $\mathcal{R}_2$) describe the effect of the threads in the environment of τ_1 (resp. τ_2); e.g. as the environment of τ_1 comprises τ_2 only and $\mathcal{G}_2$ describes the effect of τ_2 on the shared state, we have $\mathcal{R}_1 \triangleq \mathcal{G}_2$.

Let us first consider analysing τ_2 in isolation, ignoring the // annotations for now (these become clear once we present the CASL proof rules in §3). Recall that in order to detect the use-after-free bug at L, thread τ_2 must account for an interleaving in which τ_1 executes both its instructions before τ_2 proceeds with its execution. That is, τ_2 may *assume* that τ_1 executes the actions associated with α_1 and α_2, as defined in $\mathcal{R}_2$. Note that after each environment action (in $\mathcal{R}_2$) we extend the trace to record the associated action (we elaborate on why this is needed below): starting from the empty trace $[]$, we subsequently update it to $[\alpha_1]$ and $[\alpha_1, \alpha_2]$ to record the environment actions assumed to have executed. Thread τ_2 then executes the (local) assignment instruction $a := 0$ (line 7) which accesses its local state $(a \Mapsto v_a)$ only. Subsequently, it proceeds to execute its instructions by accessing/updating the shared state as prescribed in $\mathcal{G}_2$: it 1) takes action α_1' associated with executing $a := [z]$, whereby it reads from the heap location pointed to by z (i.e. l_z) and stores it in a; and then 2) takes action α_2' associated with executing $[x] := 1$, where it attempts to write to location l_x pointed to by x and arrives at a use-after-free error as l_x is deallocated, yielding $Q_2 \triangleq a \Mapsto 1 * \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto 1}$. Note that after each $\mathcal{G}_2$ action α the trace is extended with α, culminating in trace θ (defined at the top of Fig. 1). That is, each time a thread accesses the *shared* state it must do so through an action in its guarantee and record it in its trace. By contrast, when the instruction effect is limited to its *local* state (e.g. line 7 of τ_2), it may be executed freely, without consulting the guarantee or recording an action.

We next analyse τ_1 in isolation: τ_1 executes its two instructions as given by α_1 and α_2 in $\mathcal{G}_1$, updating the trace to $[\alpha_1, \alpha_2]$. It then assumes that τ_2 in its environment executes its actions (in $\mathcal{R}_1$), resulting in θ and yielding $Q_1 \triangleq \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto 1}$. Note that τ_1 may assume that the environment action α_2' executes *erroneously*, as described in $\mathcal{R}_1(\alpha_2')$.

Finally, we reason about the full program using the CASL *parallel composition* rule, Par (in Fig. 3), stating that if we prove $\mathcal{R}_1, \mathcal{G}_1, \Theta_1 \vdash [P_1]\ \mathsf{C}_1\ [\epsilon : Q_1]$ and separately $\mathcal{R}_2, \mathcal{G}_2, \Theta_2 \vdash [P_2]\ \mathsf{C}_2\ [\epsilon : Q_2]$, then we can prove $\mathcal{R}_1 \cap \mathcal{R}_2, \mathcal{G}_1 \cup \mathcal{G}_2, \Theta_1 \cap \Theta_2 \vdash [P_1 * P_2]\ \mathsf{C}_1 \,||\, \mathsf{C}_2\ [\epsilon : Q_1 * Q_2]$ for the concurrent program $\mathsf{C}_1 \,||\, \mathsf{C}_2$. In other words, (1) the pre-condition (resp. post-conditions) of $\mathsf{C}_1 \,||\, \mathsf{C}_2$ is given by composing the pre-conditions (resp. post-conditions) of its constituent threads, namely $P_1 * P_2$ (resp. $Q_1 * Q_2$); (2) the effect of $\mathsf{C}_1 \,||\, \mathsf{C}_2$ on the shared state is the union of their respective effect (i.e. $\mathcal{G}_1 \cup \mathcal{G}_2$); (3) the effect of the $\mathsf{C}_1 \,||\, \mathsf{C}_2$ environment on the shared state is the effect of the threads in the environment of both C_1 and C_2 (i.e. $\mathcal{R}_1 \cap \mathcal{R}_2$); and (4) the traces generated by $\mathsf{C}_1 \,||\, \mathsf{C}_2$ are those generated by both C_1 and C_2 (i.e. $\Theta_1 \cap \Theta_2$).

Returning to Fig. 1, we use Par to reason about the full program. Let C_1 and C_2 denote the programs in the left and right threads, respectively. (1) Starting from $P \triangleq a \Mapsto v_a * \boxed{x \Mapsto l_x * l_x \mapsto v_x * z \Mapsto l_z * l_z \mapsto v_z}$, we split P as $P_1 * P_2$ (i.e. $P = P_1 * P_2$) and pass P_1 (resp. P_2) to τ_1 (resp. τ_2). (2) We analyse C_1 and C_2 in isolation and derive $\mathcal{R}_1, \mathcal{G}_1, \{\theta\} \vdash [P_1]\ \mathsf{C}_1\ [er : Q_1]$ and $\mathcal{R}_2, \mathcal{G}_2, \{\theta\} \vdash [P_2]\ \mathsf{C}_2\ [er : Q_2]$. (3) We use Par to combine the two triples and derive $\emptyset, \mathcal{G}_1 \cup \mathcal{G}_2, \{\theta\} \vdash [P]\ \mathsf{C}_1 \,||\, \mathsf{C}_2\ [er : Q]$ with $Q \triangleq a \Mapsto 1 * \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto 1}$.

CISL versus CASL. In contrast to CISL-Par where we can only derive normal (ok) triples (and thus inevitably must encode erroneous behaviours as normal ones if possible), the CASL Par rule makes no such stipulation ($\epsilon = ok$ or $\epsilon \in \text{ErExit}$) and allows deriving both normal *and* erroneous triples. More significantly, a CISL triple $[P]\ \mathsf{C}\ [\epsilon : Q]$ executed by a thread τ only allows τ to take actions (updating the state) by executing C, i.e. only allows actions executed by τ itself and not those of other threads in the environment (executing another program C'). This is also the case for all *correctness* triples in over-approximate settings, e.g. RGSep and RG. By contrast, CASL triples additionally allow τ to *take a particular action by an environment thread*, as specified by rely, thereby allowing one to consider a specific interleaving (see the EnvL, EnvR and EnvEr rules in Fig. 3). This ability to *assume a specific execution by the environment* is missing from CISL. This is a crucial insight for data-dependent bugs that depend on certain data exchange/synchronisation between threads.

Recording Traces. Note that when taking a thread action (e.g. at line 1 in Fig. 1), the executing thread τ must adhere to the behaviour in its guarantee *and* additionally witness the action taken by executing corresponding instructions; this is captured by the CASL Atom rule. That is, the guarantee denotes what τ *can* do, and provides no assurance that τ does carry out those actions. This assurance is witnessed by executing corresponding instructions, e.g. τ_1 in Fig. 1 must execute $\mathsf{free}(x)$ on line 1 when taking α_1. By contrast, when τ takes an environment action (e.g. at line 3 in Fig. 1), it simply assumes the environment will take this action without witnessing it. That is, when reasoning about τ in isolation we *assume a particular interleaving* and show a given world is reachable under that interleaving. Therefore, the correctness of the compositional reasoning is contingent on the environment fulfilling this assumption by adhering to the *same interleaving*. This is indeed why we record θ, i.e. to ensure all threads assume the same sequence of actions on the shared state. As mentioned above, $\mathcal{R}, \mathcal{G}$ specify how the *shared* state is manipulated, and have no bearing on *thread-local* states. As such, we record no trace actions for instructions that only manipulate the local state (e.g. line 7 in Fig. 1); this is captured by the CASL AtomLocal rule.

Note that the Θ component of CASL is absent in its over-approximate counterpart RGSep. This is because in the *correctness* setting of RGSep one must prove a program is correct for *all interleavings* and it is not needed to record the interleavings considered. By contrast, in the *incorrectness* setting of CASL our aim is to show the occurrence of a bug under *certain interleavings* and thus we record them to ensure their feasibility: if a thread assumes a given interleaving θ, we must ensure that θ is a feasible interleaving for all concurrent threads.

RGSep versus RG. We develop CASL as an under-approximate analogue of RGSep [20] rather than RG [11]. We initially developed CASL as an under-approximate analogue of RG; however, the lack of support for local reasoning led to rather verbose proofs. Specifically, as discussed above and as we show in §4, the CASL AtomLocal rule allows local reasoning on thread-local resources without accounting for them in the recorded traces. By contrast, in RG there is no thread-local state and the entire state is shared (accessible by all threads). Hence, were we to base CASL on RG, we could only support the Atom rule and not the local AtomLocal variant, and thus every single action by each thread would have to be recorded in the trace. This not only leads to verbose proofs (with long traces), but it is also somewhat counter-intuitive. Specifically, thread-local computations (e.g. on thread-local registers) have no bearing on the behaviour of other threads and need not be reflected in the global trace. We present our original RG-based development [19, §E and §F] for the interested reader.

2.2 CASL for Compositional Exploit Detection

In practice, software attacks attempt to escalate privileges (e.g. Log4j) or steal credentials (e.g. Heartbleed [8]) using an *adversarial* program written by a security expert. That is, attackers typically use an adversarial program to interact with a codebase and exploit its vulnerabilities. Therefore, we can model a vulnerable program $\mathsf{C_v}$ and its adversary (attacker) $\mathsf{C_a}$ as the *concurrent* program $\mathsf{C_a} \,||\, \mathsf{C_v}$, and use CASL to detect vulnerabilities in $\mathsf{C_v}$. Vulnerabilities often fall into the *data-dependent* category, where the vulnerable program $\mathsf{C_v}$ receives an input from the adversary $\mathsf{C_a}$, and that input determines the next steps in the execution of $\mathsf{C_v}$, i.e. $\mathsf{C_a}$ affects the control flow of $\mathsf{C_v}$. Hence, existing under-approximate techniques such as CISL cannot detect such exploits, while the compositional techniques of CASL for detecting data-dependent bugs is ideally-suited for them. Indeed, to our knowledge CASL is the *first* formal, under-approximate theory that enables exploit detection. Thanks to the compositional nature of CASL, the approaches described here can be used to build *scalable* tools for exploit detection, as we discuss below. Moreover, by virtue of its under-approximate nature and built-in *no-false-positives* theorem, exploits detected by CASL are *certified* in that they are guaranteed to reveal true vulnerabilities.

In what follows we present an example of an information disclosure attack. Later we show how we use CASL to detect several classes of exploits, including: 1) *information disclosure attacks* on stacks (§4) and 2) heaps in the technical appendix [19, §C] to uncover sensitive data, e.g. Heartbleed [8]; and 3) *memory safety attacks* [19, §D], e.g. zero allocation [21].

Hereafter, we write $\mathsf{C_a}$ and $\mathsf{C_v}$ for the adversarial and vulnerable programs, respectively; and write τ_a and τ_v for the threads running $\mathsf{C_a}$ and $\mathsf{C_v}$, respectively. We represent exploits as $\mathsf{C_a} \,||\, \mathsf{C_v}$, positioning $\mathsf{C_a}$ and $\mathsf{C_v}$ as the left and right threads, respectively. As we discuss below, we model communication between τ_a and τ_v over a *shared channel* c, where each party can transmit (send/receive) information over c using the send and recv instructions.

```
                || local $sec := *$;
                || local $w[8] := \{0\}$;
send$(c, 8)$;   || recv$(c, x)$;
recv$(c, y)$;   || if $(x \leq 8)$            (InfDis)
                ||     $z := w[x]$;
                ||     send$(c, z)$;
```

Information Disclosure Attacks. Consider the InfDis example on the right, where τ_v (the vulnerable thread) allocates two variables on the stack: *sec*, denoting a secret initialised with a non-deterministic value ($*$), and array w of size 8 initialised to 0. As per stack allocation, *sec* and w are allocated *contiguously* from the top of the stack. That is, when the top of the stack is denoted by top, then *sec* occupies the first unitof the stack (at top) and w occupies the next 8 units (between top-1 and top-8). In other words, w starts at top-8 and thus $w[i]$ resides at top$-8+i$.

The τ_v then receives x from τ_a, retrieves the x^{th} entry in w and sends it to τ_a over c. Specifically, τ_v first checks that x is valid (within bounds) via $x \leq 8$. However, as arrays are indexed from 0, for x to be valid we must have $x < 8$ instead, and thus this check is insufficient. That is, when τ_a sends 8 over c (send$(c, 8)$), then τ_v receives 8 on c and stores it in x (recv(c, x)), i.e. $x{=}8$, resulting in an out-of-bounds access ($z := w[x]$). As such, since $w[i]$ resides at top$-8+i$, $x{=}8$ and *sec* is at top, accessing $w[x]$ inadvertently retrieves the secret value *sec*, stores it in z, which is subsequently sent to τ_a over c, disclosing *sec* to τ_a!

CASL for Scalable Exploit Detection. In the over-approximate setting proving *correctness* (absence of bugs), a key challenge of developing *scalable* analysis tools lies in the need to consider *all* possible interleavings and establish bug freedom for all interleavings. In the under-approximate setting proving *incorrectness* (presence of bugs), this task is somewhat easier: it suffices to find *some* buggy interleaving. Nonetheless, in the absence of heuristics guiding the search for buggy interleavings, one must examine each interleaving to find buggy ones. Therefore, in the worst case one may have to consider all interleavings.

When using CASL to detect data-dependent bugs, the problem of identifying buggy interleavings amounts to determining *when* to account for environment actions. For instance, detecting the bug in Fig. 1 relied on accounting for the actions of the left thread at lines 5 and 6 prior to reading from z. Therefore, the scalability of a CASL-based bug detection tool hinges on developing heuristics that determine when to apply environment actions.

In the general case, where all threads may access any and all shared data (e.g. in DataDep), developing such heuristics may require sophisticated analysis of the synchronisation patterns used. However, in the case of exploits (e.g. in InfDis), the adversary and the vulnerable programs operate on mostly separate states, with the shared state comprising a shared channel (c) only, accessed through send and recv. In other words, the program *syntax* (send and recv instructions) provides a simple heuristic prescribing when the environment takes an action. Specifically, the computation carried out by τ_v is mostly *local* and does not affect the shared state c (i.e. by instructions other than send/recv); as discussed, such local steps need not be reflected in the trace and τ_a need not account for them. Moreover, when τ_v encounters a recv$(c, -)$ instruction, it must first assume the environment (τ_a) takes an action and sends a message over c to be subsequently received by τ_{v}. This leads to a *simple heuristic*: take an environment action prior to executing recv. We believe this observation can pave the way towards scalable exploit detection, underpinned by CASL and benefiting from its no-false-positives guarantee, certifying that the exploits detected are true positives.

3 CASL: A General Framework for Bug Detection

We present the general theory of the CASL framework for detecting concurrency bugs. We develop CASL in a *parametric* fashion, in that CASL may be instantiated for detecting bugs and exploits in a multitude of contexts. CASL is instantiated by supplying it with the specified parameters; the soundness of the instantiated CASL reasoning is then guaranteed *for free* from the soundness of the framework (see Theorem 2). We present the CASL ingredients as well as the parameters it is to be supplied with upon instantiation.

CASL Programming Language. The CASL language is parametrised by a set of *atoms*, $\textsc{Atom}$, ranged over by $\mathbf{a}$. For instance, our CASL instance for detecting memory safety bugs [19, §D] includes atoms for accessing the heap. This allows us to instantiate CASL for different scenarios without changing its underlying meta-theory. Our language is given by the C grammar below, and includes atoms ($\mathbf{a}$), skip, sequential composition ($\mathsf{C}_1; \mathsf{C}_2$), non-deterministic choice ($\mathsf{C}_1 + \mathsf{C}_2$), loops ($\mathsf{C}^\star$) and parallel composition ($\mathsf{C}_1 \,||\, \mathsf{C}_2$).

$$\textsc{Comm} \ni \mathsf{C} ::= \mathbf{a} \mid \mathsf{skip} \mid \mathsf{C}_1; \mathsf{C}_2 \mid \mathsf{C}_1 + \mathsf{C}_2 \mid \mathsf{C}^\star \mid \mathsf{C}_1 \,||\, \mathsf{C}_2$$

CASL States and Worlds. Reasoning frameworks [12, 18] typically reason at the level of high-level states, equipped with additional instrumentation to support diverse reasoning principles. In the frameworks based on separation logic, high-level states are modelled by a *partial commutative monoid* (PCM) of the form $(\textsc{State}, \circ, \textsc{State}_0)$, where $\textsc{State}$ denotes the set of *states*; $\circ : \textsc{State} \times \textsc{State} \rightharpoonup \textsc{State}$ denotes the partial, commutative and associative *state composition function*; and $\textsc{State}_0 \subseteq \textsc{State}$ denotes the set of unit states. Two states $l_1, l_2 \in \textsc{State}$ are *compatible*, written $l_1 \# l_2$, if their composition is defined: $l_1 \# l_2 \overset{\text{def}}{\iff} \exists l.\ l{=}l_1 \circ l_2$. Once CASL is instantiated with the desired state PCM, we define the notion of *worlds*, $\textsc{World}$, comprising pairs of states of the form (l, g), where $l \in \textsc{State}$ is the *local state* accessible only by the current thread(s), and $g \in \textsc{State}$ is the *shared* (global) state accessible by all threads (including those in the environment), provided that (l, g) is *well-formed.* A pair (l, g) is well-formed if the local and shared states are compatible ($l \# g$).

▶ **Definition 1** (Worlds). *Assume a* PCM for states, $(\textsc{State}, \circ, \textsc{State}_0)$. *The set of* worlds *is* $\textsc{World} \triangleq \left\{(l, g) \in \textsc{State} \times \textsc{State} \,\middle|\, l \# g\right\}$. World composition, $\bullet : \textsc{World} \times \textsc{World} \rightharpoonup \textsc{World}$, *is defined component-wise,* $\bullet \triangleq (\circ, \circ_=)$*, where* $g \circ_= g' \triangleq g$ *when* $g{=}g'$*, and is otherwise undefined. The* world unit set *is* $\textsc{World}_0 \triangleq \left\{(l_0, g) \in \textsc{World} \,\middle|\, l_0 \in \textsc{State}_0 \wedge g \in \textsc{State}\right\}$.

Notation. We use p, q, r as metavariables for state sets (in $\mathcal{P}(\textsc{State})$), and P, Q, R as metavariables for world sets (in $\mathcal{P}(\textsc{World})$). We write $P * Q$ for $\left\{w \bullet w' \,\middle|\, w \in P \wedge w' \in Q\right\}$; $P \wedge Q$ for $P \cap Q$; $P \vee Q$ for $P \cup Q$; false for $\emptyset$; and true for $\mathcal{P}(\textsc{World})$. We write $p * \boxed{q}$ for $\left\{(l, g) \in \textsc{World} \,\middle|\, l \in p \wedge g \in q\right\}$. When clear from the context, we lift p, q, r to sets of worlds with arbitrary shared states; e.g. p denotes a set of worlds (l, g), where $l \in p$ and $g \in \textsc{State}$.

$$\alpha \in \textsc{AId} \qquad \mathcal{R},\mathcal{G} \in \textsc{AMap} \triangleq \textsc{AId} \rightharpoonup \mathcal{P}(\textsc{State}) \times \textsc{Exit} \times \mathcal{P}(\textsc{State}) \qquad \Theta \in \mathcal{P}(\textsc{Trace})$$

$$\theta \in \textsc{Trace} \triangleq \textsc{List}\langle \textsc{AId} \rangle \qquad \Theta_0 \triangleq \{[\,]\} \qquad \Theta_1 +\!\!+ \Theta_2 \triangleq \left\{ \theta_1 +\!\!+ \theta_2 \,\middle|\, \theta_1 \in \Theta_1 \land \theta_2 \in \Theta_2 \right\}$$

$$\alpha :: \Theta \triangleq \left\{ \alpha :: \theta \,\middle|\, \theta \in \Theta \right\} \qquad \mathsf{dsj}(\mathcal{R},\mathcal{G}) \overset{\text{def}}{\iff} dom(\mathcal{R}) \cap dom(\mathcal{G}) = \emptyset$$

$$\mathcal{R}_1 \subseteq \mathcal{R}_2 \overset{\text{def}}{\iff} dom(\mathcal{R}_1) \subseteq dom(\mathcal{R}_2) \land \forall \alpha \in dom(\mathcal{R}_1).\, \mathcal{R}_1(\alpha) = \mathcal{R}_2(\alpha)$$

$$\mathcal{R}' \preccurlyeq_\theta \mathcal{R} \overset{\text{def}}{\iff} \forall \alpha \in \theta \cap dom(\mathcal{R}').\, \mathcal{R}'(\alpha) = \mathcal{R}(\alpha) \qquad \mathcal{R}' \preccurlyeq_\Theta \mathcal{R} \overset{\text{def}}{\iff} \forall \theta \in \Theta.\, \mathcal{R}' \preccurlyeq_\theta \mathcal{R}$$

$$\mathsf{wf}(\mathcal{R},\mathcal{G}) \overset{\text{def}}{\iff} \mathsf{dsj}(\mathcal{R},\mathcal{G}) \land \forall \alpha \in dom(\mathcal{R}), p, q, l.\, \mathcal{R}(\alpha) = (p, -, q) \land q * \{l\} \neq \emptyset \Rightarrow p * \{l\} \neq \emptyset$$

Figure 2 The CASL model definitions.

Error Conditions and Atomic Axioms. CASL uses under-approximate triples [16, 17, 18] of the form $\mathcal{R},\mathcal{G},\Theta \vdash [p]\ \mathsf{C}\ [\epsilon : q]$, where $\epsilon \in \textsc{Exit} \triangleq \{ok\} \uplus \textsc{ErExit}$ denotes an *exit condition*, indicating normal (ok) or erroneous execution ($\epsilon \in \textsc{ErExit}$). Erroneous conditions in $\textsc{ErExit}$ are reasoning-specific and are supplied as a parameter, e.g. npe for a null pointer exception.

We shortly define the under-approximate proof system of CASL. As atoms are a CASL parameter, the CASL proof system is accordingly parametrised by their set of under-approximate *axioms*, $\textsc{Axiom} \subseteq \mathcal{P}(\textsc{State}) \times \textsc{Atom} \times \textsc{Exit} \times \mathcal{P}(\textsc{State})$, describing how they may update states. Concretely, an atomic axiom is a tuple $(p, \mathbf{a}, \epsilon, q)$, where $p, q \in \mathcal{P}(\textsc{State})$, $\mathbf{a} \in \textsc{Atom}$ and $\epsilon \in \textsc{Exit}$. As we describe shortly, atomic axioms are then lifted to CASL proof rules (see Atom and AtomLocal), describing how atomic commands may modify worlds.

CASL Triples. A CASL triple $\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}\ [\epsilon : Q]$ states that every world in Q can be reached under ϵ for every *witness trace* $\theta \in \Theta$ by executing C on some world in P. Moreover, at each step the actions of the current thread (executing C) and its environment adhere to $\mathcal{G}$ and $\mathcal{R}$, respectively. The $\mathcal{R},\mathcal{G}$ are defined as *action maps* in Fig. 2, mapping each action $\alpha \in \textsc{AId}$ to a triple describing its behaviour. Compared to original rely/guarantee relations [20, 11], in CASL we record two additional components: 1) the exit condition (ϵ) indicating a normal or erroneous step; and 2) the action id (α) to identify actions uniquely. The latter allows us to construct a witness interleaving $\theta \in \textsc{Trace}$ as a list of actions (see Fig. 2). As discussed in §2, to avoid false positives, if we detect a bug assuming the environment takes action α, we must indeed witness the environment taking α. That is, if we detect a bug assuming the environment takes α but the environment cannot do so, then the bug is a false positive. Recording traces ensures each thread fulfils its assumptions, as we describe shortly.

Intuitively, each α corresponds to executing an atom that updates a *sub-part* of the shared state. Specifically, $\mathcal{G}(\alpha) = (p, \epsilon, q)$ (resp. $\mathcal{R}(\alpha) = (p, \epsilon, q)$) denotes that the current thread (resp. an environment thread) may take α and update a shared sub-state in p to one in q under ϵ, and in doing so it extends each trace in Θ with α. Moreover, the current thread may take α with $\mathcal{G}(\alpha) = (p, \epsilon, q)$ only if it executes an atom $\mathbf{a}$ with behaviour (p, ϵ, q), i.e. $(p, \mathbf{a}, \epsilon, q) \in \textsc{Axiom}$, thereby *witnessing* α. By contrast, this is not required for an environment action. As we describe below, this is because each thread witnesses the $\mathcal{G}$ actions it takes, and thus when combining threads (using the CASL Par rule described below), so long as they agree on the interleavings (traces) taken, then the actions recorded have been witnessed.

Lastly, we require $\mathcal{R}$, $\mathcal{G}$ to be *well-formed* ($\mathsf{wf}(\mathcal{R}, \mathcal{G})$ in Fig. 2), stipulating that: 1) $\mathcal{R}$ and $\mathcal{G}$ be *disjoint*, $\mathsf{dsj}(\mathcal{R}, \mathcal{G})$; and 2) the actions in $\mathcal{R}$ be *frame-preserving*: for all α with $\mathcal{R}(\alpha) = (p, -, q)$ and all states l, if l is compatible with q (i.e. $q * \{l\} \neq \emptyset$), then l is also compatible with p (i.e. $p * \{l\} \neq \emptyset$). Condition (1) allows us to attribute actions uniquely to threads (i.e. distinguish between $\mathcal{R}$ and $\mathcal{G}$ actions). Condition (2) is necessary for the CASL Frame rule (see below), ensuring that applying an environment action does not inadvertently update the state in such a way that invalidates the resources in the frame. Note that we require no such condition on $\mathcal{G}$ actions. This is because as discussed, each $\mathcal{G}$ action taken is witnessed by executing an atom axiomatised in Axiom; axioms in Axiom must in turn be frame-preserving to ensure the soundness of CASL. That is, a $\mathcal{G}$ action is taken only if it is witnessed by an atom which is frame-preserving by definition (see SoundAtoms in [19, §A]).

CASL Proof Rules. We present the CASL proof rules in Fig. 3, where we assume the rely/guarantee relations in triple contexts are well-formed. Skip states that executing skip leaves the worlds (P) unchanged and takes no actions, yielding a single empty trace $\Theta_0 \triangleq \{[\,]\}$. Seq, SeqEr, Choice, Loop1, Loop2 and BackwardsVariant are analogous to those of IL [16] with $S : \mathbb{N} \to \mathcal{P}(\text{World})$. Note that in Seq, the set of traces resulting from executing $\mathsf{C}_1; \mathsf{C}_2$ is given by $\Theta_1 {+\!\!+} \Theta_2$ (defined in Fig. 2) by point-wise combining the traces of C_1 and C_2.

Atom describes how executing an atom $\mathbf{a}$ affects the shared state: when the local state is in p' and the shared state is in $p * f$, i.e. a sub-part of the shared state is in p, then executing $\mathbf{a}$ with $(p' * p, \mathbf{a}, \epsilon, q' * q) \in \text{Axiom}$ updates the local state from p' to q' and the shared sub-part from p to q, provided that the effect on the shared state is given by a guarantee action α ($\mathcal{G}(\alpha) = (p, \epsilon, q)$). That is, the $\mathcal{G}$ action only captures the shared state, and the thread may update its local state freely. In doing so, we *witness* α and record it in the set of traces ($\{[\alpha]\}$). By contrast, AtomLocal states that so long as executing $\mathbf{a}$ does not touch the shared state, it may update the local state arbitrarily, without recording an action.

EnvL, EnvR and EnvEr are the Atom counterparts in that they describe how the *environment* may update the shared state. Specifically, EnvL and EnvR state that the current thread may be interleaved by the environment. Given $\alpha \in \mathit{dom}(\mathcal{R})$, the current thread may execute C either *after* or *before* the environment takes action α, as captured by EnvL and EnvR, respectively. In the case of EnvL we further require that α (in $\mathit{dom}(\mathcal{R})$) denote a normal ($\mathit{ok}$) execution step, as otherwise the execution would short-circuit and the current thread could not execute C. Note that unlike in Atom, the environment action α in EnvL and EnvR only updates the shared state; e.g. in EnvL the p sub-part of the shared state is updated to r and the local state p' is left unchanged. Analogously, EnvEr states that executing C may terminate erroneously under er if it is interleaved by an *erroneous* step of the environment under er. That is, if the environment takes an erroneous step, the execution of the current thread is terminated, as per the short-circuiting semantics of errors.

Note that Atom ensures action α is taken by the current thread (in $\mathcal{G}$) only when the thread witnesses it by executing a matching atom. By contrast, in EnvL, EnvR and EnvEr we merely *assume* the environment takes action α in $\mathcal{R}$. As such, each thread locally ensures that it takes the guarantee actions in its traces.As shown in Par, when joining the threads via parallel composition $\mathsf{C}_1 \,||\, \mathsf{C}_2$, we ensure their sets of traces agree: $\Theta_1 \cap \Theta_2 \neq \emptyset$. Moreover, to ensure we can attribute each action in traces to a unique thread, we require that $\mathcal{G}_1$ and $\mathcal{G}_2$ be disjoint ($\mathsf{dsj}(\mathcal{G}_1, \mathcal{G}_2)$, see Fig. 2). Finally, when τ_1 and τ_2 respectively denote the threads running C_1 and C_2, the $\mathcal{R}_1 \subseteq \mathcal{G}_2 \cup \mathcal{R}_2$ premise ensures when τ_1 attributes an action α to $\mathcal{R}_1$ (i.e. α is in $\mathcal{R}_1$), then α is an action of either τ_2 (i.e. α is in $\mathcal{G}_2$) or its environment (i.e. of a thread running concurrently with both τ_1 and τ_2); similarly for $\mathcal{R}_2 \subseteq \mathcal{G}_1 \cup \mathcal{R}_1$.

$$\textsc{Skip}\quad \mathcal{R},\mathcal{G},\Theta_0 \vdash [P]\ \mathsf{skip}\ [ok\colon P]$$

$$\textsc{Seq}\quad \frac{\mathcal{R},\mathcal{G},\Theta_1 \vdash [P]\ \mathsf{C}_1\ [ok\colon R] \qquad \mathcal{R},\mathcal{G},\Theta_2 \vdash [R]\ \mathsf{C}_2\ [\epsilon : Q]}{\mathcal{R},\mathcal{G},\Theta_1 +\!\!+\ \Theta_2 \vdash [P]\ \mathsf{C}_1;\mathsf{C}_2\ [\epsilon : Q]}$$

$$\textsc{SeqEr}\quad \frac{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}_1\ [er\colon Q] \qquad er \in \textsc{ErExit}}{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}_1;\mathsf{C}_2\ [er\colon Q]}$$

$$\textsc{Atom}\quad \frac{\mathcal{G}(\alpha)=(p,\epsilon,q) \qquad (p' * p, \mathbf{a}, \epsilon, q' * q) \in \textsc{Axiom}}{\mathcal{R},\mathcal{G},\{[\alpha]\} \vdash [p' * \boxed{p * f}]\ \mathbf{a}\ [\epsilon : q' * \boxed{q * f}]}$$

$$\textsc{Loop1}\quad \mathcal{R},\mathcal{G},\Theta_0 \vdash [P]\ \mathsf{C}^\star\ [ok\colon P]$$

$$\textsc{Loop2}\quad \frac{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}^\star;\mathsf{C}\ [\epsilon : Q]}{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}^\star\ [\epsilon : Q]}$$

$$\textsc{AtomLocal}\quad \frac{(p,\mathbf{a},ok,q) \in \textsc{Axiom}}{\mathcal{R},\mathcal{G},\{[\,]\} \vdash [p]\ \mathbf{a}\ [ok\colon q]}$$

$$\textsc{BackwardsVariant}\quad \frac{\forall k.\ \mathcal{R},\mathcal{G},\Theta \vdash [S(k)]\,\mathsf{C}\,[ok\colon S(k{+}1)] \qquad \forall n{>}0.\ \Theta_n = \Theta +\!\!+\ \Theta_{n-1}}{\mathcal{R},\mathcal{G},\Theta_n \vdash [S(0)]\ \mathsf{C}\ [ok\colon S(n)]}$$

$$\textsc{Choice}\quad \frac{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}_i\ [\epsilon{:}Q] \quad \text{for some } i \in \{1,2\}}{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}_1 + \mathsf{C}_2\ [\epsilon : Q]}$$

$$\textsc{Comb}\quad \frac{\mathcal{R},\mathcal{G},\Theta_1 \vdash [P]\ \mathsf{C}\ [\epsilon{:}Q] \qquad \mathcal{R},\mathcal{G},\Theta_2 \vdash [P]\ \mathsf{C}\ [\epsilon{:}Q]}{\mathcal{R},\mathcal{G},\Theta_1 \cup \Theta_2 \vdash [P]\ \mathsf{C}\ [\epsilon : Q]}$$

$$\textsc{EnvL}\quad \frac{\mathcal{R}(\alpha)=(p,ok,r) \qquad \mathcal{R},\mathcal{G},\Theta \vdash [p' * \boxed{r * f}]\ \mathsf{C}\ [\epsilon{:}Q]}{\mathcal{R},\mathcal{G},\alpha :: \Theta \vdash [p' * \boxed{p * f}]\ \mathsf{C}\ [\epsilon : Q]}$$

$$\textsc{EnvR}\quad \frac{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}\ [ok{:}r' * \boxed{r * f}] \qquad \mathcal{R}(\alpha)=(r,\epsilon,q)}{\mathcal{R},\mathcal{G},\Theta +\!\!+\ \{[\alpha]\} \vdash [P]\ \mathsf{C}\ [\epsilon : r' * \boxed{q * f}]}$$

$$\textsc{EnvEr}\quad \frac{\mathcal{R}(\alpha) = (p, er, q) \qquad er \in \textsc{ErExit}}{\mathcal{R},\mathcal{G},\{[\alpha]\} \vdash [\boxed{p * f}]\ \mathsf{C}\ [er\colon \boxed{q * f}]}$$

$$\textsc{Frame}\quad \frac{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}\ [\epsilon : Q] \qquad \mathsf{stable}(R, \mathcal{R} \cup \mathcal{G})}{\mathcal{R},\mathcal{G},\Theta \vdash [P * R]\ \mathsf{C}\ [\epsilon : Q * R]}$$

$$\textsc{ParEr}\quad \frac{\begin{array}{c}\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}_i\ [er\colon Q] \quad \text{for some } i \in \{1,2\} \\ er \in \textsc{ErExit} \qquad \Theta \sqsubseteq \mathcal{G}\end{array}}{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}_1 \,||\, \mathsf{C}_2\ [er\colon Q]}$$

$$\textsc{Cons}\quad \frac{\begin{array}{c}P' \subseteq P \qquad \mathcal{R}',\mathcal{G}',\Theta' \vdash [P']\ \mathsf{C}\ [\epsilon : Q'] \qquad Q \subseteq Q' \\ \mathcal{R} \preccurlyeq_\Theta \mathcal{R}' \qquad \mathcal{G} \preccurlyeq_\Theta \mathcal{G}' \qquad \Theta \subseteq \Theta'\end{array}}{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}\ [\epsilon : Q]}$$

$$\textsc{Par}\quad \frac{\begin{array}{c}\mathcal{R}_1,\mathcal{G}_1,\Theta_1 \vdash [P_1]\ \mathsf{C}_1\ [\epsilon{:}Q_1] \qquad \mathcal{R}_2,\mathcal{G}_2,\Theta_2 \vdash [P_2]\ \mathsf{C}_2\ [\epsilon{:}Q_2] \\ \mathcal{R}_1 \subseteq \mathcal{G}_2 \cup \mathcal{R}_2 \qquad \mathcal{R}_2 \subseteq \mathcal{G}_1 \cup \mathcal{R}_1 \qquad \mathsf{dsj}(\mathcal{G}_1,\mathcal{G}_2) \qquad \Theta_1 \cap \Theta_2 \neq \emptyset\end{array}}{\mathcal{R}_1 \cap \mathcal{R}_2, \mathcal{G}_1 \cup \mathcal{G}_2, \Theta_1 \cap \Theta_2 \vdash [P_1 * P_2]\ \mathsf{C}_1 \,||\, \mathsf{C}_2\ [\epsilon : Q_1 * Q_2]}$$

$$\begin{array}{lrcl}\text{with} & \Theta \sqsubseteq \mathcal{G} & \overset{\text{def}}{\iff} & \forall \theta \in \Theta.\ \theta \subseteq dom(\mathcal{G}) \\ \text{and} & \mathsf{stable}(R,\mathcal{R}) & \overset{\text{def}}{\iff} & \forall (l,g) \in R, \alpha.\ \forall (p,-,q) \in \mathcal{R}(\alpha), g_q \in q, g_p \in p, g'.\ g = g_q \circ g' \Rightarrow (l, g_p \circ g') \in R\end{array}$$

Figure 3 The CASL proof rules, where $\mathcal{R}/\mathcal{G}$ relations in contexts are well-formed.

Observe that Par can be used for both normal and erroneous triples (i.e. for any ϵ) *compositionally*. This is in contrast to CISL, where only ok triples can be proved using CISL-Par, and thus bugs can be detected only if they can be encoded as ok (see §2). In other words, CISL cannot compositionally detect either data-agnostic bugs with short-circuiting semantics or data-dependent bugs altogether, while CASL can detect both data-agnostic and data-dependent bugs compositionally using Par, without the need to encode them as ok. This is because CASL captures the environment in $\mathcal{R}$, enabling compositional reasoning. In particular, even when we do not know the program in parallel, so long as its behaviour adheres to $\mathcal{R}$, we can detect an error: $\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}\ [er\colon Q]$ ensures the error is reachable as long as the environment adheres to $\mathcal{R}$, without knowing the program run in parallel to C.

ParEr is the concurrent analogue of SeqEr, describing the short-circuiting semantics of concurrent executions: given $i \in \{1, 2\}$, if running C_i in isolation results in an error, then running $\mathsf{C}_1 \,||\, \mathsf{C}_2$ also yields an error. The $\Theta \sqsubseteq \mathcal{G}$ premise (defined in Fig. 3) ensures the actions in Θ are from $\mathcal{G}$, i.e. taken by the current thread and not assumed to have been taken by the environment. Comb allows us to extend the traces: if the traces in both Θ_1 and Θ_2 witness the execution of C, then the traces in $\Theta_1 \cup \Theta_2$ also witness the execution of C.

Cons is the CASL rule of consequence. As with under-approximate logics [16, 17, 18], the post-worlds Q may shrink ($Q \subseteq Q'$) and the pre-worlds P may grow ($P' \subseteq P$). The traces may shrink ($\Theta \subseteq \Theta'$): if traces in Θ' witness executing C, then so do the traces in the smaller set Θ. Lastly, $\mathcal{R} \preccurlyeq_\Theta \mathcal{R}'$ (resp. $\mathcal{G} \preccurlyeq_\Theta \mathcal{G}'$) defined in Fig. 2 states that the rely (resp. guarantee) may *grow or shrink* so long as it preserves the behaviour of actions in Θ. This is in contrast to RG/RGSep where the rely may only shrink and the guarantee may only grow. This is because in RG/RGSep one must defensively prove correctness against *all* environment actions at *all program points*, i.e. for *all interleavings*. Therefore, if a program is correct under a larger environment (with more actions) $\mathcal{R}'$, then it is also correct under a smaller environment $\mathcal{R}$. In CASL, however, we show an outcome is reachable under a set of witness interleavings Θ. Hence, for traces in Θ to remain valid witnesses, the rely/guarantee may grow or shrink, so long as they faithfully reflect the behaviours of the actions in Θ.

Lastly, Frame states that if we show $\mathcal{R}, \mathcal{G}, \Theta \vdash [P]\, \mathsf{C}\, [\epsilon : Q]$, we can also show $\mathcal{R}, \mathcal{G}, \Theta \vdash [P * R]\, \mathsf{C}\, [\epsilon : Q * R]$, so long as the worlds in R are *stable* under $\mathcal{R}, \mathcal{G}$ ($\mathsf{stable}(R, \mathcal{R} \cup \mathcal{G})$, defined in Fig. 3), in that R accounts for possible updates. That is, given $(l, g) \in R$ and α with $(p, -, q) \in \mathcal{R}(\alpha) \cup \mathcal{G}(\alpha)$, if a sub-part g_q of the shared g is in q ($g = g_q \circ g'$ for some $g_q \in q$ and g'), then replacing g_q with an arbitrary $g_p \in p$ results in a world (i.e. $(l, g_p \circ g')$) also in R.

CASL Soundness. We define the formal interpretation of CASL triples via *semantic triples* of the form $\mathcal{R}, \mathcal{G}, \Theta \models [P]\ \mathsf{C}\ [\epsilon : Q]$ (see [19, §A]). We show CASL is sound by showing its triples in Fig. 3 induce valid semantics triples. We do this in the theorem below, with its proof in [19, §B].

▶ **Theorem 2** (Soundness). *For all $\mathcal{R}, \mathcal{G}, \Theta, p, \mathsf{C}, \epsilon, q$, if $\mathcal{R}, \mathcal{G}, \Theta \vdash [p]\ \mathsf{C}\ [\epsilon : q]$ is derivable using the rules in Fig. 3, then $\mathcal{R}, \mathcal{G}, \Theta \models [p]\ \mathsf{C}\ [\epsilon : q]$ holds.*

4 CASL for Exploit Detection

We present CASL$_{\mathsf{ID}}$, a CASL instance for detecting *stack-based information disclosure* exploits. In the technical appendix [19] we present CASL$_{\mathsf{HID}}$ for detecting *heap-based information disclosure* exploits [19, §C] and CASL$_{\mathsf{MS}}$ for detecting *memory safety attacks* [19, §D].

The CASL$_{\mathsf{ID}}$ atomics, Atom$_{\mathsf{ID}}$, are below, where $\mathtt{L} \in \mathbb{N}$ is a label, x, y are (local) variables, c is a shared channel and v is a value. They include assume statements and primitives for generating a random value $*$ ($\mathsf{local}\, x :=_\tau *$) used to model a secret value (e.g. a private key), declaring an array x of size n initialised with v ($\mathsf{local}\, x[n] :=_\tau \{v\}$), array assignment $\mathtt{L}\colon x[k] :=_\tau y$, sending ($\mathsf{send}(c, x)$ and $\mathsf{send}(c, v)$) and receiving ($\mathsf{recv}(c, x)$) over channel c. As is standard, we encode $\mathsf{if}\ (b)\ \mathsf{then}\ \mathsf{C}_1\ \mathsf{else}\ \mathsf{C}_2$ as $(\mathsf{assume}(b); \mathsf{C}_1) + (\mathsf{assume}(\neg b); \mathsf{C}_2)$.

$$
\begin{aligned}
\text{Atom}_{\mathsf{ID}} \ni \mathbf{a} ::= {} & \mathtt{L}\colon \mathsf{assume}(b) \mid \mathtt{L}\colon \mathsf{local}\ x :=_\tau\ * \mid \mathtt{L}\colon \mathsf{local}\ x[k] :=_\tau\ \{v\} \mid \mathtt{L}\colon x :=_\tau y[k] \\
& \mid \mathtt{L}\colon \mathsf{send}(c, x)_\tau \mid \mathtt{L}\colon \mathsf{send}(c, v)_\tau \mid \mathtt{L}\colon \mathsf{recv}(c, x)_\tau
\end{aligned}
$$

CASL$_{\mathsf{ID}}$ States. A CASL$_{\mathsf{ID}}$ state, $(s, h, \mathbf{h})$, comprises a *variable stack* $s \in \text{Stack} \triangleq \text{Var} \rightharpoonup \widetilde{\text{Val}}$, mapping variables to *instrumented values*; a *heap* $h \in \text{Heap} \triangleq \text{Loc} \rightharpoonup (\widetilde{\text{Val}} \cup \text{List}\langle\widetilde{\text{Val}}\rangle)$, mapping shared locations (e.g. channel c) to (lists of) instrumented values; and a *ghost*

ID-VarSecret
$$[\mathbf{s}_\tau \dashrightarrow n]\ \text{L: local}\ x :=_\tau *\ [ok\colon \mathbf{s}_\tau \dashrightarrow (n{+}1) * x = \mathsf{top}-n * x \Mapsto (v,\tau,1)]$$

ID-VarArray
$$[\mathbf{s}_\tau \dashrightarrow n * k>0]\ \text{L: local}\ x[k] :=_\tau \{v\}\left[ok\colon \mathbf{s}_\tau \dashrightarrow (n{+}k) * x = \mathsf{top}-(n{+}k{-}1) * \mathop{\circledast}_{j=0}^{k-1}(x{+}j \Mapsto (v,\tau,0)) * k>0\right]$$

ID-ReadArray
$$[k \Mapsto (v,\tau_v,b) * y{+}v \Mapsto V_y * x \Mapsto -]\ \text{L: } x :=_\tau y[k]\ [ok\colon k \Mapsto (v,\tau_v,b) * y{+}v \Mapsto V_y * x \Mapsto V_y]$$

ID-SendVal
$$[c \mapsto L]\ \text{L: send}(c,v)_\tau\ [ok\colon c \mapsto L +\!\!+ [(v,\tau,0)]]$$

ID-Send
$$[c \mapsto L * x \Mapsto V]\ \text{L: send}(c,x)_\tau\ [ok\colon c \mapsto L +\!\!+ [V]]$$

ID-Recv
$$[c \mapsto [(v,\tau_t,\iota)] +\!\!+ L * x \Mapsto - * (\iota=0 \lor \tau \in \mathsf{Trust})]\ \text{L: recv}(c,x)_\tau\ [ok\colon c \mapsto L * x \Mapsto (v,\tau_t,\iota) * (\iota=0 \lor \tau \in \mathsf{Trust})]$$

ID-RecvEr
$$[c \mapsto [(v,\tau_t,1)] +\!\!+ L * \tau \notin \mathsf{Trust}]\ \text{L: recv}(c,x)_\tau\ [er\colon c \mapsto [(v,\tau_t,1)] +\!\!+ L * \tau \notin \mathsf{Trust}]$$

Figure 4 The CASL$_{\text{ID}}$ axioms.

heap $\mathbf{h} \in \text{GHeap} \triangleq (\{\mathbf{s}\} \times \text{TId}) \rightharpoonup \text{Val}$, tracking the stack size ($\mathbf{s}$). An instrumented value, $(v,\tau,\iota) \in \widetilde{\text{Val}} \triangleq \text{Val} \times \text{TId} \times \{0,1\}$, comprises a value ($v$), its provenance ($\tau$, the thread from which v originated), and its *secret attribute* ($\iota \in \{0,1\}$) denoting whether the value is secret (1) or not (0). We use x, y as metavariables for local variables, c for shared channels, v for values, L for value lists and V for instrumented values. State composition is defined as $(\uplus,\uplus,\uplus)$, where $\uplus$ denotes disjoint function union. The state unit set is $\{(\emptyset,\emptyset,\emptyset)\}$. We write $x \Mapsto V$ for states in which the stack comprises a single variable x mapped on to V and the heap and ghost heaps are empty, i.e. $\{([x \mapsto V],\emptyset,\emptyset)\}$. Similarly, we write $c \mapsto L$ for $\{(\emptyset,[c \mapsto L],\emptyset)\}$, and $\mathbf{s}_\tau \dashrightarrow v$ for $\{(\emptyset,\emptyset,[(\mathbf{s},\tau)\mapsto v])\}$.

CASL$_{\text{ID}}$ Axioms. We present the CASL$_{\text{ID}}$ atomic axioms in Fig. 4. We assume that each variable declaration (via local $x :=_\tau *$ and local $x[n] :=_\tau \{v\}$) defines a *fresh* name, and thus avoid the need for variable renaming at declaration time. We assume the stack top is given by the constant top; thus when the stack of thread τ is of size n (i.e. $\mathbf{s}_\tau \dashrightarrow n$), the next empty stack spot is at $\mathsf{top}-n$. Executing L: local $x :=_\tau *$ in ID-VarSecret increments the stack size ($\mathbf{s}_\tau \dashrightarrow n{+}1$), reserves the next empty spot for x and initialises x with a value (v) marked secret (1) with its provenance (thread τ). Analogously, ID-VarArray describes declaring an array of size k, where the next k spots are reserved for x (the $\circledast$ denotes $*$-iteration: $\circledast_{j=1}^{n}(x{+}j \Mapsto V) \triangleq x{+}1 \Mapsto V * \cdots * x{+}n \Mapsto V$). When k holds value v, ID-ReadArray reads the v^{th} entry of y (at $y{+}v$) in x. ID-SendVal extends the content of c with $(v,\tau,0)$. ID-Recv describes *safe* data receipt (not leading to *information disclosure*), i.e. the value received is not secret ($\iota=0$) or the recipient is *trusted* ($\tau \in \mathsf{Trust} \triangleq \text{TId} \setminus \{\tau_{\mathsf{a}}\}$). By contrast, ID-RecvEr describes when receiving data leads to information disclosure, i.e. the value received is secret and the recipient is untrusted ($\tau \notin \mathsf{Trust}$), in which case the state is unchanged.

Example: InfDis. In Fig. 5 we present a CASL$_{\text{ID}}$ proof sketch of the information disclosure exploit in InfDis. The proof of the full program is given in Fig. 5a. Starting from $P_a * P_v$ with a singleton empty trace (Θ_0, defined in Fig. 2), we use Par to pass P_a and P_v respectively to τ_{a} and τ_{v}, analyse each thread in isolation, and combine their results (Q_a and Q_v) into $Q_a * Q_v$, with the two agreeing on the trace set Θ generated. Figures 5b and 5c show the proofs of τ_{a} and τ_{v}, respectively, where we have also defined their pre- and post-conditions.

$$\mathcal{R}_v(\alpha'_1) \triangleq (c \mapsto [\,], ok, c \mapsto [(n, \tau_a, 0)]) \qquad \mathcal{R}_v(\alpha'_2) \triangleq (c \mapsto [(v, \tau, 1)], ok, c \mapsto [\,]) \qquad \mathcal{R}_a \triangleq \mathcal{G}_v \qquad \mathcal{G}_a \triangleq \mathcal{R}_v$$
$$\mathcal{G}_v(\alpha_1) \triangleq (c \mapsto [(n, \tau_a, 0)], ok, c \mapsto [\,]) \qquad \mathcal{G}_v(\alpha_2) \triangleq (c \mapsto [\,], ok, c \mapsto (v, \tau, 1)) \qquad \Theta \triangleq \{[\alpha'_1, \alpha_1, \alpha_2, \alpha'_2]\}$$

$$\emptyset, \mathcal{G}_a \cup \mathcal{G}_v, \Theta_0 \vdash [P_a * P_v] \quad // \text{PAR}$$
$$\begin{array}{l||l}
 & \mathcal{R}_v, \mathcal{G}_v, \Theta_0 \vdash [P_v] \\
 & \quad \text{L}_1\text{: local } sec :=_{\tau_v} * \\
\mathcal{R}_a, \mathcal{G}_a, \Theta_0 \vdash [P_a] & \quad \text{L}_2\text{: local } w[8] :=_{\tau_v} \{v\} \\
\quad \text{L}'_1\text{: send}(c, 8)_{\tau_a} & \quad \text{L}_3\text{: recv}(c, x)_{\tau_v} \\
\quad \text{L}'_2\text{: recv}(c, y)_{\tau_a} & \quad \text{L}_4\text{: } z :=_{\tau_v} w[x] \\
\mathcal{R}_a, \mathcal{G}_a, \Theta \vdash [er\colon Q_a] & \quad \text{L}_5\text{: send}(c, z)_{\tau_v} \\
 & \mathcal{R}_v, \mathcal{G}_v, \Theta \vdash [er\colon Q_v]
\end{array}$$
$$\emptyset, \mathcal{G}_a \cup \mathcal{G}_v, \Theta \vdash [er\colon Q_a * Q_v]$$

(a)

$$\mathcal{R}_a, \mathcal{G}_a, \Theta_0 \vdash \left[P_a \triangleq \boxed{c \mapsto [\,]} * \tau_a \notin \text{Trust}\right]$$
$$\quad \text{L}'_1\text{: send}(c, 8)_{\tau_a} \quad // \text{ATOM} + \text{ID-SENDVAL}$$
$$\mathcal{R}_a, \mathcal{G}_a, \{[\alpha'_1]\} \vdash \left[ok\colon \boxed{c \mapsto [(8, \tau_a, 0)]} * \tau_a \notin \text{Trust}\right]$$
$$// \text{ENVL} \times 2$$
$$\mathcal{R}_a, \mathcal{G}_a, \{[\alpha'_1, \alpha_1, \alpha_2]\} \vdash \left[ok\colon \boxed{c \mapsto [(v, \tau_v, 1)]} * \tau_a \notin \text{Trust}\right]$$
$$\quad \text{L}'_2\text{: recv}(c, t)_{\tau_a} \quad // \text{ATOM} + \text{ID-RECVER}$$
$$\mathcal{R}_a, \mathcal{G}_a, \Theta \vdash \left[er\colon Q_a \triangleq \boxed{c \mapsto [(v, \tau_v, 1)]} * \tau_a \notin \text{Trust}\right]$$

(b)

$$\mathcal{R}_v, \mathcal{G}_v,$$
$$\Theta_0 \vdash \left[P \triangleq \mathbf{s}_{\tau_v} \dashrightarrow 0 * x \Mapsto - * z \Mapsto - * \boxed{c \mapsto [\,]}\right]$$
$$\quad \text{L}_1\text{: local } sec :=_{\tau_v} * \quad // \text{ATOMLOCAL+ID-VARSECRET}$$
$$\Theta_0 \vdash \left[ok\colon \mathbf{s}_{\tau_v} \dashrightarrow 1 * x \Mapsto - * z \Mapsto - * \boxed{c \mapsto [\,]} * sec{=}\text{top} * sec \Mapsto (v_s, \tau_v, 1)\right]$$
$$\quad \text{L}_2\text{: local } w[8] :=_{\tau_v} \{v\}; \quad // \text{ATOMLOCAL} + \text{ID-VARARRAY}$$
$$\Theta_0 \vdash \left[ok\colon \mathbf{s}_{\tau_v} \dashrightarrow 9 * x \Mapsto - * z \Mapsto - * \boxed{c \mapsto [\,]} * sec{=}\text{top} * sec \Mapsto (v_s, \tau_v, 1) * w{=}\text{top}-8 * \mathop{\huge *}_{j=0}^{7} (w{+}j \Mapsto (v, \tau_v))\right]$$
$$// \text{FRAME}$$
$$\Theta_0 \vdash \left[ok\colon x \Mapsto - * z \Mapsto - * \boxed{c \mapsto [\,]} * sec{=}\text{top} * sec \Mapsto (v_s, \tau_v, 1) * w{=}\text{top}-8\right] \quad // \text{ENVL}$$
$$\{[\alpha'_1]\} \vdash \left[ok\colon x \Mapsto - * z \Mapsto - * \boxed{c \mapsto [(8, \tau_a, 0)]} * sec{=}\text{top} * sec \Mapsto (v_s, \tau_v, 1) * w{=}\text{top}-8\right]$$
$$\quad \text{L}_3\text{: recv}(c, x)_{\tau_v}; \quad // (\text{ATOM} + \text{ID-RECV})$$
$$\{[\alpha'_1, \alpha_1]\} \vdash \left[ok\colon x \Mapsto (8, \tau_a, 0) * z \Mapsto - * \boxed{c \mapsto [\,]} * sec{=}\text{top} * sec \Mapsto (v_s, \tau_v, 1) * w{=}\text{top}-8\right] \quad // \text{CONS}$$
$$\{[\alpha'_1, \alpha_1]\} \vdash \left[ok\colon x \Mapsto (8, \tau_a, 0) * z \Mapsto - * \boxed{c \mapsto [\,]} * sec{=}w{+}8 * sec \Mapsto (v_s, \tau_v, 1) * w{=}\text{top}-8\right]$$
$$\quad \text{if } (x \leq 8) \quad \text{L}_4\text{: } z :=_{\tau_v} w[x] \quad // \text{ATOMLOCAL+ID-READARRAY}$$
$$\{[\alpha'_1, \alpha_1]\} \vdash \left[ok\colon x \Mapsto (8, \tau_a, 0) * z \Mapsto (v_s, \tau_v, 1) * \boxed{c \mapsto [\,]} * sec{=}w{+}8 * sec \Mapsto (v_s, \tau_v, 1) * w{=}\text{top}-8\right]$$
$$\quad \text{L}_5\text{: send}(c, z)_{\tau_v} \quad // \text{ATOM+ID-SEND}$$
$$\{[\alpha'_1, \alpha_1, \alpha_2]\} \vdash \left[ok\colon x \Mapsto (8, \tau_a, 0) * z \Mapsto (v_s, \tau_v, 1) * \boxed{c \mapsto [(v_s, \tau_v, 1)]} * sec{=}w{+}8 * sec \Mapsto (v_s, \tau_v, 1) * w{=}\text{top}-8\right]$$
$$// \text{ENVER}$$
$$\Theta \vdash \left[er\colon x \Mapsto (8, \tau_a, 0) * z \Mapsto (v_s, \tau_v, 1) * \boxed{c \mapsto [(v_s, \tau_v, 1)]} * sec{=}w{+}8 * sec \Mapsto (v_s, \tau_v, 1) * w{=}\text{top}-8\right]$$
$$\Theta \vdash \left[\begin{array}{l} er\colon Q_v \triangleq \mathbf{s}_{\tau_v} \dashrightarrow 9 * x \Mapsto (8, \tau_a, 0) * z \Mapsto (v_s, \tau_v, 1) * \boxed{c \mapsto [(v_s, \tau_v, 1)]} * sec{=}w{+}8 * sec \Mapsto (v_s, \tau_v, 1) \\ \qquad * w{=}\text{top}-8 * \mathop{\huge *}_{j=0}^{7} (w{+}j \Mapsto (v, \tau_v)) \end{array}\right]$$

(c)

Figure 5 Proofs of InfDis (a), its adversary (b) and vulnerable (c) programs.

All stack variables are local and channel c is the only shared resource. As such, rely/guarantee relations describe how τ_a and τ_v transmit data over c: α_1 and α_2 capture the recv and send in τ_v, while α'_1 and α'_2 capture the send and recv in τ_a. Using AtomLocal and CASL$_{\text{ID}}$ axioms, τ_v executes its first two instructions. It then uses Frame to frame off unneeded resources and applies EnvL to account for τ_a sending $(8, \tau_a, 0)$ over c. Using Atom with ID-Recv it receives this value in x. After using Cons to rewrite sec $= \text{top} * w = \text{top}-8$ equivalently to sec$=w+8 * w=\text{top}-8$, it applies AtomLocal with ID-ReadArray to read from $w[x]$ (i.e. the secret value at sec$=w+8$) in z. It then sends this value over c, arriving at an error using EnvEr as the value received by the adversary τ_a is secret. The last line then adds on the resources previously framed off. The proof of τ_a in Fig. 5b is analogous.

5 Related Work

Under-Approximate Reasoning. CASL builds on and generalises CISL [18]. As with IL [16] and ISL [17], CASL is an instance of under-approximate reasoning. However, IL and ISL support only sequential programs and not concurrent ones. Vanegue [22] recently developed adversarial logic (AL) as an under-approximate technique for detecting exploits. While we model C_v and C_a as $\mathsf{C}_a \,||\, \mathsf{C}_v$ as with AL, there are several differences between AL and CASL. CASL is a general, under-approximate framework that can be 1) used to detect both exploits and bugs in concurrent programs, while AL is tailored towards exploits only; 2) *instantiated* for different classes of bugs/exploits, while the model of AL is hard-coded. Moreover, CASL borrows ideas from CISL to enable 3) *state-local* reasoning (only over parts of the state accessed), while AL supports global reasoning only (over the entire state); and 4) *thread-local* reasoning (analysing each thread in isolation), while AL accounts for all threads.

Automated Exploit Generation. Determining the exploitability of bugs is central to prioritising fixes at large scale. *Automated exploit generation* (AEG) tools craft an exploit based on predetermined heuristics and preconditioned symbolic execution of unsafe binary programs [2, 5]. Additional improvements use random walk techniques to exploit heap buffer overflow vulnerabilities reachable from known bugs [9, 1, 10]. Exploits for use-after-free vulnerabilities [23] and unsafe memory write primitives [6] have also been partially automated.

As with CASL, AEG tools are fundamentally under-approximate and may not find all attacks. Assumptions made by AEG tools are hard-coded in their implementation, in contrast to CASL which can be instantiated for new classes of vulnerabilities without redesigning the core logic from scratch. Finally, traditional AEG tools have no notion of locality and require global reasoning, making existing tools unable to cope with the path explosion problem and large targets without compromising coverage. By contrast, CASL mostly acts on local states, making it more suitable for large-scale exploit detection than current tools.

References

1 Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 641–652, 2016.

2 Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

3 Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. Racerd: Compositional static race detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276514.

4 James Brotherston, Paul Brunet, Nikos Gorogiannis, and Max Kanovich. A compositional deadlock detector for android java. In *Proceedings of ASE-36*. ACM, 2021. URL: http://www0.cs.ucl.ac.uk/staff/J.Brotherston/ASE21/deadlocks.pdf.

5 Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012.

6 Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. {KOOBE}: Towards facilitating exploit generation of kernel {Out-Of-Bounds} write vulnerabilities. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1093–1110, 2020.

7 Facebook, 2021. URL: `https://fbinfer.com/`.

8 Heartbleed. The heartbleed bug, 2014. URL: `https://heartbleed.com/`.

9 Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 763–779, 2018.

10 Sean Heelan, Tom Melham, and Daniel Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1689–1706, 2019.

11 C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983. `doi:10.1145/69575.69577`.

12 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 637–650, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2676726.2676980`.

13 Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022. `doi:10.1145/3527325`.

14 Lars Müller. KPTI: A mitigation method against meltdown, 2018. URL: `https://www.cs.hs-rm.de/~kaiser/events/wamos2018/Slides/mueller.pdf`.

15 Peter W. O'Hearn. Resources, concurrency and local reasoning. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, pages 49–67, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

16 Peter W. O'Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL):10:1–10:32, December 2019. `doi:10.1145/3371078`.

17 Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. Local reasoning about the presence of bugs: Incorrectness separation logic. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 225–252, Cham, 2020. Springer International Publishing.

18 Azalea Raad, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. Concurrent incorrectness separation logic. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. `doi:10.1145/3498695`.

19 Azalea Raad, Julien Vanegue, Josh Berdine, and Peter O'Hearn. Technical appendix, 2023. URL: `https://www.soundandcomplete.org/papers/CONCUR2023/CASL/appendix.pdf`.

20 Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco T. Vasconcelos, editors, *CONCUR 2007 – Concurrency Theory*, pages 256–271, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

21 Julien Vanegue. Zero-sized heap allocations vulnerability analysis. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, WOOT'10, pages 1–8, USA, 2010. USENIX Association.

22 Julien Vanegue. Adversarial logic. *Proc. ACM Program. Lang.*, (SAS), December 2022.

23 Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. {FUZE}: Towards facilitating exploit generation for kernel {Use-After-Free} vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 781–797, 2018.