Compositional Vulnerability Detection with Insecurity Separation Logic (Extended Version)

Toby Murray[1], Pengbo Yan[1] and Gidon Ernst[2]

[1] University of Melbourne, Melbourne, Australia
toby.murray@unimelb.edu.au pengpoy@student.unimelb.edu.au
[2] LMU Munich, Munich, Germany
gidon.ernst@lmu.de

Abstract. Memory-safety issues and information leakage are known to be depressingly common. We consider the compositional static detection of these kinds of vulnerabilities in first-order C-like programs. Indeed the latter are *relational* hyper-safety violations, comparing pairs of program executions, making them more challenging to detect than the former, which require reasoning only over individual executions. Existing symbolic leakage detection methods treat only non-interactive programs, avoiding the challenges of nondeterminism. Also, being whole-program analyses they cannot be applied one-function-at-a-time, thereby ruling out incremental analysis. We remedy these shortcomings by presenting Insecurity Separation Logic (InsecSL), an under-approximate relational program logic for soundly detecting information leakage and memory-safety issues in interactive programs. Importantly, InsecSL reasons about pairs of executions, and so is relational, but purposefully resembles the non-relational Incorrectness Separation Logic (ISL) that is already automated in the Infer tool. We show how InsecSL can be automated by bi-abduction based symbolic execution, and we evaluate two implementations of this idea (one based on Infer) on various case-studies.

1 Introduction

Almost all program logics are for proving the correctness of programs. Hoare logic is a classic example, whose judgements have the form $\{P\}\ c\ \{Q\}$ for a program command c and pre- and postconditions P and Q. This judgement means that when executed from an initial state satisfying P that after command c finishes, Q is guaranteed to hold. In this sense postcondition Q *over-approximates* the final states that command c can reach from an initial P-state. Recently, interest has emerged in program logics for proving *incorrectness* [15], i.e., for diagnosing bugs in programs with a true-positives guarantee. Such logics inherit the *under-approximate* structure of Reverse Hoare Logic [8]. Their judgements $[P]\ c\ [Q]$ mean that for all final states t satisfying Q, there exists an initial P-state from which Q can execute to terminate in state t. Thus Q under-approximates the final states that command c can reach from an initial P-state.

While the two approaches are roughly equivalent for deterministic programs, under-approximate reasoning is necessary to accurately diagnose vulnerabilities

in *nondeterministic* programs, including those that allocate memory or interact with an outside environment or user. Incorrectness Separation Logic [16,12] (ISL) is such an under-approximate logic, which has proved especially useful for automatic memory-safety bug detection because program analysis in the logic can be carried out automatically via bi-abduction based symbolic execution [5,16], and supports compositional and incremental program analysis [12].

All such under-approximate logics to-date, however, reason only about individual program executions. They can therefore detect only those bugs that can be observed in this way, like assertion failures (as in Incorrectness Logic [15]) or memory-safety errors like null-pointer dereferences and use-after-free errors (as in Incorrectness Separation Logic [16]). Yet, vulnerabilities come in many kinds, beyond memory-safety issues. In this paper we focus on the *automatic detection of information leakage vulnerabilities*. These are especially interesting as they are very common and can be devastating. But since information leakage is semantically expressed as a hyperproperty [6], which compares *pairs* of program executions, it is out of scope for the existing under-approximative logics.

Can we design an under-approximate logic for reasoning about such vulnerabilities which inherits the nice property that all defects which are flagged are true positives? If so, can analysis using this logic be automated to produce a compositional vulnerability analysis method?

Contribution: We answer both of these questions in the affirmative. In this paper, we present Insecurity Separation Logic (InsecSL, Section 4), an under-approximate separation logic for diagnosing information leakage and memory-safety vulnerabilities. InsecSL reasons about pairs of program executions but purposefully closely resembles the (single execution) logic ISL [16]. We show in Section 5 how reasoning in InsecSL can be automated via bi-abduction based symbolic execution by formalising and proving that the same symbolic execution procedure as is used for ISL is also sound for InsecSL. We demonstrate the practicality of our ideas by implementing them in two different tools (Section 6), including an extension of the Infer tool in which we adapt Infer's ISL implementation to diagnose information leakage vulnerabilities via InsecSL. We evaluate our implementations (Section 7) by applying them to a range of case studies. Soundness theorems (namely Theorem 1 for InsecSL and Theorem 2 for symbolic execution respectively) have been mechanised in Isabelle/HOL. All artifacts are available online: `https://covern.org/insecurity.html`.

2 Motivation

We use the program in Fig. 1 to both motivate and explain our approach. This program implements the core of a simple *sealed-bid* auction server. In a sealed-bid auction, all information about bids must be kept secret until after the auction is finished, at which point only the winning bid is announced.

Bids in this auction are pairs of `int`s: (id, qt) where id identifies the bidder who submitted the bid, and qt is the amount (or *quote*) submitted in the bid. The C struct type `bid_t` pairs these two values together. The top-level function

```
struct bid_t { int id; int qt; };

void run_auction() {
  struct bid_t highest = /* init */;
  while (/* still going */) {
    struct bid_t bid;
    get_bid(&bid);
    update_max(&highest, &bid);
  }
  announce_winner(&highest);
}
```

```
void update_max(struct bid_t *a,
                struct bid_t *b)
{
  /* branching on secrets: */
  if (b->qt > a->qt) {
    a->id = b->id;
    a->qt = b->qt;
    /* potentially slow: */
    log_current_max(a->id, a->qt);
  }
}
```

Fig. 1. The core of a sealed-bid auction server, adapted from a case-study in SecC: https://bitbucket.org/covern/secc/src/master/examples/case-studies/auction.c.

`run_auction()` maintains the current maximum bid `highest`, and a temporary `bid` used to store newly submitted bids, which are received via the `get_bid()` function. Each new bid is then compared to the current highest one using the function `update_max()`, which potentially updates the current highest bid and persists a record about this fact via `log_current_max`. Note that `get_bid()` is inherently nondeterministic: It may return arbitrary values, since it is the interface between the program and its environment. This puts it outside the scope of Relational Symbolic Execution [11] as implemented in tools like Binsec/Rel [7].

Unfortunately, `update_max()` is insecure. As it updates the maximum bid only when the newly submitted bid is larger than the current maximum, its *timing* depends on whether the branch is taken or not. This timing leak can be exploited by auction participants to game the auction. In particular if `log_current_max` incurs a notable delay—writing to disk or even network storage synchronously may be slow—they might be capable to infer whether the bid they have submitted is greater than the current maximum or not. Moreover, the call to `announce_winner()` is potentially insecure under the premise that we only want to disclose the winning bid. If `highest` has not been computed correctly, then we may accidentally reveal sensitive information about another bid.

Challenge: The question of whether a *potential* information leak in a program becomes critical therefore strongly depends on the context in which functions like `update_max()` and `announce_winner()` are called. A compositional underapproximative analysis like that of InsecSL must therefore be capable of tracking such relationships *precisely* to be sound, i.e., to avoid false positives.

As an example, the security-related summary inferred for `update_max()`, shown below, expresses that each potentially insecure final state as marked by *insec* is guaranteed to be reachable under the sufficient presumption that parameters `a` and `b` are valid pointers. Assertion $(bqt > aqt) \not\sim \ell$ denotes that this insecurity occurs if within a given calling context the outcome of the conditional $bqt > aqt$ is

not already known to the attacker of security level ℓ (cf. Section 3 and Section 4).

$$[\texttt{\&b->qt} \mapsto bqt * \texttt{\&a->qt} \mapsto aqt]$$
$$\texttt{update\_max(a,b)}$$
$$[insec\colon (bqt > aqt) \not\preceq \ell * \texttt{\&b->qt} \mapsto bqt * \texttt{\&a->qt} \mapsto aqt]$$

Note that this summary is beyond the scope of type systems like [18] which just capture whether information flow happens or not, but which fail to adequately reason about logical conditions like $(bqt > aqt) \not\preceq \ell$.

3 Attacker Model

We imagine that the execution of the program in question is being observed by an *attacker*, who has certain observational powers and initial knowledge and is trying to deduce secret information that the program is trying to protect. An information leak occurs if the attacker can deduce some secret information that they did not already know initially before the program was executed.

As standard, the attacker is assumed to know the program being executed and certain initial values in memory as specified by assertions characterising pre-states. The program may perform inputs and outputs during its execution and the attacker is assumed to be able to observe some of these. All other information is considered *secret*, and information flow security requires that the attacker can never learn any new information above that which they were assumed to know initially. As usual, we therefore define what an attacker can observe with the help of a security lattice comprised of labels ℓ which are comparable by a binary relation $\sqsubseteq$ with **low** and **high** being the least resp. greatest elements, modeling public and fully sensitive information, respectively. A channel at level ℓ' is observable by an ℓ-attacker if $\ell' \sqsubseteq \ell$, e.g., the **low** channel is observable publicly.

As motivated in Section 2, the security property for InsecSL is *timing-sensitive*. This means that the attacker can not just observe inputs and outputs on certain channels, but also at what times they occur. As is typical, time is measured in terms of the number of small-steps of execution in the language's small-step operational semantics. Following the standard *program counter (PC) security model* [13], the security property targeted by InsecSL assumes an attacker who is able to observe at each point in time (i.e. after each small-step of the semantics) the program code that is running. This implies that e.g. when executing an if-conditions **if** e **then** c_1 **else** c_2 **endif** where $c_1 \neq c_2$, that the attacker can infer some information about e (namely whether it evaluated to true or not), since they will be able to tell in the subsequent execution step whether c_1 or c_2 is being executed. A similar argument applies to while-loops. While not as strong as *constant-time security* [3], InsecSL can be easily extended to cover the stronger attacker model of constant-time security if desired (see Appendix A.3).

We emphasize that the choice of this attacker model is a trade-off: under this attacker model it is not possible to verify programs that have if/while conditions

that depend on secrets, even if leakage from such conditions is considered acceptable in certain situations. On the other hand, a PC-security security guarantee requires one to consider only "matched" executions, as exploited by SecCSL [10] and also by InsecSL, which drastically simplifies the logic and its automation in comparison to product constructions like [9].

4 Insecurity Separation Logic (InsecSL)

Insecurity Separation Logic (InsecSL) is the relational analogue of ISL [16] and the underapproximative dual to Security (Concurrent) Separation Logic (SecCSL) [10]. Judgements in InsecSL are written as

$$\vdash_\ell [P]\ c\ [\epsilon\colon\ Q] \tag{1}$$

where relational assertions P characterizes the pre-states ("presumption") and Q characterize reachable final states ("result"), ℓ is a security level, c is a program command, and ϵ is a status flag that indicates whether the command has terminated normally ($\epsilon = \mathit{ok}$), whether a runtime error has occurred ($\epsilon = \mathit{err}(L)$), or whether an insecurity has been detected ($\epsilon = \mathit{insec}(L)$). The latter two track a program location L that points to the cause of the defect.

The capability to precisely characterise insecurity for nondeterministic programs is what distinguishes InsecSL from prior logics. As an example, InsecSL allows us to derive that the output of the value of an expression e to a channel of security level ℓ' can be potentially witnessed as insecure without further presumptions in any (pair of final) state(s) in which e is secret wrt. ℓ', written $e \not\preceq \ell'$, under the assumption of an ℓ-attacker (which implies $\ell' \sqsubseteq \ell$):

$$\frac{}{\vdash_\ell [\mathbf{emp}]\ L\colon \mathbf{output}(\ell', e)\ [\mathit{insec}(L)\colon e \not\preceq \ell']}\ \textsc{OutInsec} \tag{2}$$

Judgement (1) is defined relative to a relational semantics of assertions like $e \not\preceq \ell'$ and **emp**, written $(s,h)\ (s',h') \models_\ell P$ where s, s' are stores (mappings from variables to values) and h, h' are heaps (mappings from addresses to values), and a small-step program semantics $k_1 \xrightarrow{\sigma} k_2$ where configurations k are either a running program $k_1, k_2 = \langle \mathbf{run}\ \text{"}c\text{"}\ s\ h\rangle$, a terminated execution $k_2 = \langle \mathbf{stop}\ s\ h\rangle$ or a program error $k_2 = \langle \mathbf{abort}\ s\ h\rangle$, where the latter two correspond to a final status ϵ of ok and $\mathit{err}(L)$, respectively.

As a hyperproperty, security cannot be defined solely by looking at the final state of a single execution, comprised of the store s and heap h in $\langle \mathbf{stop}\ s\ h\rangle$ configurations. Instead, we have to compare what is *observable* between possible pairs of executions. To capture this notion, execution steps additionally keep track of relevant events as a schedule σ, which records for example input events $\mathsf{in}\langle \ell', v\rangle$ and outputs events $\mathsf{out}\langle \ell', v\rangle$ to track a value v together with the security level ℓ' of the respective communication channel. The key issue for defining a security logic like InsecSL (and also SecCSL) and proving soundness of rules like (2) is therefore to connect the three ingredients, namely the judgements (1), observations σ, and the assertions P, Q encountered throughout a derivation. It is based on the following semantic notion:

Definition 1 (Execution Witness). *Presumption P and result Q witness an execution of program c against the ℓ-level attacker and a given status ϵ when for all final states s, h, s', h' such that (s,h) $(s',h') \vDash_\ell Q$, there exist initial states s_0, h_0, s_0', h_0', and σ, σ', k, k' such that (s_0,h_0) $(s_0',h_0') \vDash_\ell P$ and $\langle$**run** "c" s_0 $h_0\rangle \xrightarrow{\sigma}{}^* k$ and $\langle$**run** "c" s_0' $h_0'\rangle \xrightarrow{\sigma'}{}^* k'$, where σ and σ' have equal lengths and are* input-equivalent *for the ℓ-level attacker (Definition 2), and the final store and heap of k are respectively s and h and likewise for k', s' and h'. Moreover,*

If $\epsilon = ok$ resp. $\epsilon = err(L)$ then
- *σ and σ' are* output-equivalent *to the ℓ-level attacker (Definition 2),*
- *and k and k' must both be* **stop***ped resp.* **abort***ed.*

If $\epsilon = insec(L)$ then
- *either σ and σ' are* not *output-equivalent to the ℓ-level attacker,*
- *or k and k' both denote* **run***ning configurations with different commands.*

Witnessing an insecure behaviour therefore violates the standard security condition of program counter (PC) security [13]. Also note that the conditions are mutually exclusive, i.e., an execution witness can uniquely be classified into an ok behavior, an erroneous behavior, or an insecure one.

Theorem 1 (True Positives). InsecSL *guarantees that if $\vdash_\ell [P]\ c\ [\epsilon\colon\ Q]$ is derivable via the rules, shown in Fig. 2, then there is an execution witness for P, Q, c, and ϵ wrt. an ℓ-attacker, according to Definition 1.*

Assertions. InsecSL assertions are *relational* [19,10]; pure assertions ρ and spatial assertions P, Q are defined according to the following grammar:

$$\begin{array}{rcl} \rho & ::= & e \mid \rho \Longrightarrow \rho \mid e :: e_\ell \mid e \not:: e_\ell \\ P, Q & ::= & \mathbf{emp} \mid \rho \mid e \mapsto e' \mid e \not\mapsto \mid P * Q \mid \exists x.\ P \mid P \Longrightarrow Q \end{array}$$

where e ranges over pure expressions, including boolean propositions (first case of ρ), similarly, e_ℓ ranges over pure expression that denote security labels of some designated data type that models the security lattice and includes constants **low** and **high** but is not further specified here.

Semantically, assertions are evaluated over *pairs* of states, written $s\ s' \models_\ell \rho$ and (s,h) $(s',h) \models_\ell P$ for stores s, s' and heaps h, h', where the unprimed resp. primed states come from the two executions being compared. Stores are mappings from variable names to values as usual, whereas heaps $h\colon Val \rightharpoonup Val \cup \{\bot\}$ are partial functions that include an additional $\bot$ element as in ISL, where $p \in dom(h)$ and $h(p) = \bot$ means that pointer p is definitely invalid in contrast to $p \notin dom(h)$, which means we do not currently have access resp. own p.

The key definitions are as follows (see Fig. 5 for the full list):

$$s\ s' \vDash_\ell e \iff [e]_s = \mathbf{true} \wedge [e]_{s'} = \mathbf{true} \tag{3}$$

$$s\ s' \vDash_\ell e :: e_\ell \iff [e_\ell]_s \sqsubseteq \ell \wedge [e_\ell]_{s'} \sqsubseteq \ell \implies [e]_s = [e]_{s'} \tag{4}$$

$$s\ s' \vDash_\ell e \not:: e_\ell \iff [e_\ell]_s \sqsubseteq \ell \wedge [e_\ell]_s \sqsubseteq \ell \wedge [e]_s \neq [e]_{s'} \tag{5}$$

where we define $(s,h)\ (s',h') \models \rho$ iff $s\ s \models \rho$ and $h = h' = \emptyset$, and $[e]_s$ denotes the evaluation of pure expression e in store s, and $\sqsubseteq$ is the partial order between security labels. Conditions $[e_\ell]_s \sqsubseteq \ell$ and $[e_\ell]_{s'} \sqsubseteq \ell$ therefore mean that e_ℓ denotes a security label that is relevant wrt. the "current" ℓ-attacker from $\models_\ell$ resp. (1).

We can assert a pure boolean expression e if it is known to hold in both states s and s' (3). Assertion $e :: e_\ell$ denotes *agreement* of value e with respect to the security label denoted by e_ℓ, i.e., the value of e is the same in both s and s' (4). It coincides with $\mathbb{A}\, e$ of [2] for $e_\ell = \mathbf{low}$ but just as in SecCSL [10], e_ℓ can be a more complex expression, not just a constant. It expresses that an e_ℓ-attacker knows the value of e, specifically $e :: \mathbf{low}$ means that e is public. Dually, *disagreement* $e \not:: e_\ell$ formalises that an attacker who can observe level e_ℓ has some uncertainty about e (5). Semantically, $s, s' \models_\ell e \not:: e_\ell$ requires that it is possible for the expression e to take two *different* values in the two stores s and s' being compared. Therefore, leaking the value of e to an e_ℓ-visible output channel is insecure because the attacker can learn whether the system is actually in state s or in s' by observing the value of e.

The second feature for bug-detection is the assertion $e \not\mapsto$ from ISL [16], which expresses that e is known to be an invalid pointer, so that dereferencing e is necessarily incorrect. This is dual to the standard points-to assertion $e \mapsto e'$ which states that memory location e is valid and contains value e'.

We point out that relational implication $\Longrightarrow$ is distinct from pure implication at the level of expressions (not shown here). All other connectives intuitively mean the same as in a non-relational setting, e.g., **emp** denotes an empty heap and $P * Q$ asserts P and Q on two disjoint parts of the heap, but of course technically these have to be lifted to the relational setting semantically.

Commands and Semantics. Commands c in the language are as follows, where e is a pure expression that can mention program variables x:

$$\begin{aligned} c \ ::= \ & \mathbf{skip} \mid x := e \mid x := [e] \mid [e] := e' \mid x := \mathbf{alloc}(e) \mid \mathbf{free}(e) \mid \\ & L\colon c \mid c_1; c_2 \mid \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{endif} \mid \mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{done} \mid \\ & \mathbf{output}(e, e') \mid x := \mathbf{input}(e) \end{aligned}$$

Here $[e]$ denotes dereferencing pointer e and e.g. in C would be written $*\mathtt{e}$. As in ISL [16], commands in InsecSL carry an optional label L that is used for error-reporting, written $L\colon c$. Most commands are standard, except $x := \mathbf{input}(e_\ell)$ and $\mathbf{output}(e_\ell, e)$. Command $x := \mathbf{input}(e_\ell)$ means input a value from the channel denoted by e_ℓ and assign the inputted value to the variable x; command $\mathbf{output}(e_\ell, e)$ means to output the value denoted by the expression e on the output channel denoted by the expression e_ℓ.

The language of InsecSL is given a small-step semantics $k_1 \xrightarrow{\sigma} k_2$, allowing judgements to talk about partial executions ending in **run**ning non-final states (cf. *insec*(L) case in Definition 1). Importantly, this semantics records the values and security labels of input and output commands as part of schedule σ, which is necessary to state the formal security properties used for InsecSL's soundness result in Theorem 1 via Definition 2 below.

The schedule is a list of events $e ::= \tau \mid \mathsf{in}\langle \ell, v\rangle \mid \mathsf{out}\langle \ell, v\rangle \mid \mathsf{allocate}\langle v\rangle$ for security level ℓ and value $v \in \mathit{Val}$. Event τ represents a single, non-**input**, non-**output**, non-**alloc** step of computation, i.e., τ steps are not critical for security. Event $\mathsf{in}\langle \ell, v\rangle$ records that value v was input at security level ℓ and $\mathsf{out}\langle \ell, v\rangle$ records that value v was output at security level (i.e. on the output channel) ℓ, while $\mathsf{allocate}\langle v\rangle$ records that address v was dynamically allocated. It is simply included as a convenience to ensure that all non-determinism can be resolved by the schedule σ. Some key rules are shown below, the full listing is in Fig. 4.

$$\frac{a = [p]_s \qquad h(a) = v}{\langle \textbf{run } \text{“}x := [p]\text{”}\ s\ h\rangle \xrightarrow{[\tau]} \langle \textbf{stop } s(x := v)\ h\rangle}$$

$$\frac{a = [p]_s \qquad a \notin \mathit{dom}(h) \vee h(a) = \bot}{\langle \textbf{run } \text{“}x := [p]\text{”}\ s\ h\rangle \xrightarrow{[\tau]} \langle \textbf{abort } s\ h\rangle}$$

$$\langle \textbf{run } \text{“}x := \textbf{input}(e_\ell)\text{”}\ s\ h\rangle \xrightarrow{[\mathsf{in}\langle [e_\ell]_s, v\rangle]} \langle \textbf{stop } s(x := v)\ h\rangle$$

$$\langle \textbf{run } \text{“}\textbf{output}(e_\ell, e)\text{”}\ s\ h\rangle \xrightarrow{\mathsf{out}\langle [e_\ell]_s, [e]_s\rangle} \langle \textbf{stop } s\ h\rangle$$

The first rule shows a load via pointer expression p from a valid address a, the corresponding value in the heap is then assigned to variable x in the updated store $s(x := v)$. Notice that we can observe memory errors in this semantics directly by transitions to $\langle \textbf{abort } s\ h\rangle$ configurations, as it is for example when the pointer expression p instead evaluates to an unknown address $a \notin \mathit{dom}(h)$ or one that is definitely not allocated $h(a) = \bot$ (second rule). Reading from an input channel returns a non-deterministic value v that is assigned to x in the successor state. However, information leakage can only be observed by comparing pairs of executions in terms of their schedules (cf. Definition 1).

As an example, $\textbf{output}(e_\ell, e)$ with $[e_\ell]_s = \ell'$ and $[e_\ell]_{s'} = \ell'$ in a pair of executions with stores s and s' respectively, will expose two schedules $\sigma = [\mathsf{out}\langle \ell', v\rangle]$ and $\sigma' = [\mathsf{out}\langle \ell', v'\rangle]$, where $v = [e]_s$ and $v' = [e]_{s'}$ are the values that are output over the channel in the two runs. If $\ell' \sqsubseteq \ell$, i.e., the channel is visible to the attacker, then an information leak occurs if $v \neq v'$ and we have an execution witness according to Definition 1 and Eq. (5) for result $[\mathit{insec}\colon e \not\sim \ell']$. Input- and output-equivalence which Definition 1 relies on is therefore as follows:

Definition 2 (Input and Output Equivalence). *Two schedules are are* input *resp.* output equivalent *for the ℓ-level attacker when all inputs resp. outputs observable to that attacker are identical in each, i.e., after projecting the schedules to those input resp. output events, $\mathsf{in}\langle \ell', v\rangle$ or $\mathsf{out}\langle \ell', v\rangle$ for which $\ell' \sqsubseteq \ell$.*

Proof Rules and Soundness. The proof rules of InsecSL are in Fig. 2. Rules analog to those of ISL [16] are included, those rules that mention value classification (e.g. in Input) and those with insec result are specific to InsecSL.

Rule LoadErr captures the case when loading via pointer p leads to an error, which is reachable from a presumption $p \not\mapsto$, i.e., states in which p is definitely an

$$\frac{}{\vdash_\ell [x = x']\ x := \mathbf{input}(e)\ [ok\colon x :: e[x'/x]]}\textsc{Input} \qquad \frac{}{\vdash_\ell [x = x']\ x := e\ [ok\colon x = e[x'/x]]}\textsc{Assign}$$

$$\frac{}{\vdash_\ell [P]\ \mathbf{skip}\ [ok\colon P]}\textsc{Skip} \qquad \frac{}{\vdash_\ell [x = x' * p \mapsto e]\ x := [p]\ [ok\colon x = e[x'/x] * p \mapsto e[x'/x]]}\textsc{LoadOK}$$

$$\frac{}{\vdash_\ell [p \mapsto e]\ [p] := e'\ [ok\colon p \mapsto e']}\textsc{StoreOK} \qquad \frac{}{\vdash_\ell [p \not\mapsto]\ L\colon x := [p]\ [err(L)\colon p \not\mapsto]}\textsc{LoadErr}$$

$$\frac{}{\vdash_\ell [p \not\mapsto]\ L\colon [p] := e\ [err(L)\colon p \not\mapsto]}\textsc{StoreErr} \qquad \frac{}{\vdash_\ell [\mathbf{emp}]\ x := \mathbf{alloc}(e)\ [ok\colon x \mapsto e]}\textsc{Alloc1}$$

$$\frac{}{\vdash_\ell [p \not\mapsto]\ x := \mathbf{alloc}(e)\ [ok\colon x = p * p \mapsto e]}\textsc{Alloc2} \qquad \frac{}{\vdash_\ell [p \mapsto e]\ \mathbf{free}(p)\ [ok\colon p \not\mapsto]}\textsc{FreeOK}$$

$$\frac{}{\vdash_\ell [p \not\mapsto]\ L\colon \mathbf{free}(p)\ [err(L)\colon p \not\mapsto]}\textsc{FreeErr} \qquad \frac{}{\vdash_\ell [\mathbf{emp}]\ \mathbf{output}(\ell', e)\ [ok\colon e :: \ell']}\textsc{OutOK}$$

$$\frac{}{\vdash_\ell [\mathbf{emp}]\ L\colon \mathbf{output}(\ell', e)\ [insec(L)\colon e \not\sim \ell']}\textsc{OutInsec}$$

$$\frac{\vdash_\ell [b * P]\ c_1\ [\epsilon\colon\ Q]}{\vdash_\ell [P]\ \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{endif}\ [\epsilon\colon\ Q]}\textsc{IfTrue} \qquad \frac{\vdash_\ell [\neg b * P]\ c_2\ [\epsilon\colon\ Q]}{\vdash_\ell [P]\ \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{endif}\ [\epsilon\colon\ Q]}\textsc{IfFalse}$$

$$\frac{c = L\colon \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{endif} \qquad c_1 \neq c_2}{\vdash_\ell [(b = \mathbf{true}) \not\sim \ell * F]\ c\ [insec(L)\colon (b = \mathbf{true}) \not\sim \ell * F]}\textsc{IfInsec}$$

$$\frac{\vdash_\ell [b * P]\ c; \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{done}\ [\epsilon\colon\ Q]}{\vdash_\ell [P]\ \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{done}\ [\epsilon\colon\ Q]}\textsc{WhileTrue}$$

$$\frac{}{\vdash_\ell [\neg b * F]\ \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{done}\ [ok\colon \neg b * F]}\textsc{WhileFalse}$$

$$\frac{}{\vdash_\ell [(b = \mathbf{true}) \not\sim \ell * F]\ L\colon \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{done}\ [insec(L)\colon (b = \mathbf{true}) \not\sim \ell * F]}\textsc{WhileInsec}$$

$$\frac{\vdash_\ell [P]\ c_1\ [ok\colon Q] \qquad \vdash_\ell [Q]\ c_2\ [\epsilon\colon\ R]}{\vdash_\ell [P]\ c_1; c_2\ [\epsilon\colon\ R]}\textsc{SeqOK} \qquad \frac{\vdash_\ell [P]\ c_1\ [err(L)\colon Q]}{\vdash_\ell [P]\ c_1; c_2\ [err(L)\colon Q]}\textsc{SeqErr}$$

$$\frac{\vdash_\ell [P]\ c_1\ [insec(L)\colon Q]}{\vdash_\ell [P]\ c_1; c_2\ [insec(L)\colon Q]}\textsc{SeqInsec} \qquad \frac{\vdash_\ell [P]\ c\ [\epsilon\colon\ Q] \qquad mod(c) \cap fv(R) = \emptyset}{\vdash_\ell [P * R]\ c\ [\epsilon\colon\ Q * R]}\textsc{Frame}$$

$$\frac{P' \overset{\ell}{\Longrightarrow} P \qquad \vdash_\ell [P']\ c\ [\epsilon\colon\ Q'] \qquad Q \overset{\ell}{\Longrightarrow} Q'}{\vdash_\ell [P]\ c\ [\epsilon\colon\ Q]}\textsc{Cons} \qquad \frac{\vdash_\ell [P_1]\ c\ [\epsilon\colon\ Q_1] \qquad \vdash_\ell [P_2]\ c\ [\epsilon\colon\ Q_2]}{\vdash_\ell [P_1 \vee P_2]\ c\ [\epsilon\colon\ Q_1 \vee Q_2]}\textsc{Disj}$$

$$\frac{\vdash_\ell [P]\ c\ [\epsilon\colon\ Q] \qquad x \notin fv(c)}{\vdash_\ell [\exists x.\ P\ x]\ c\ [\epsilon\colon\ \exists\ x.\ Q\ x]}\textsc{Ex}$$

Fig. 2. The rules of InsecSL.

invalid pointer [16]. It is formulated as a "small axiom" as typical for separation logic which is put into larger context by the standard frame rule (which is valid in our setting). We remark that sequential composition, too, works as expected.

Rule Input derives that the new value of variable x in the result can be classified with respect to e_ℓ—auxiliary variable x' is just a technical artifact to lift e over the assignment to x if e depends on x. Input commands can never be insecure, instead, manifest the domain assumption that only e_ℓ-attackers can observe the value that has been stored in x so that x is rightly classified by the level denoted by e_ℓ. Soundness of the rule therefore considers whether $x :: e_\ell[x'/x]$ holds in a given trace, i.e., whether $[x]_{s(x:=v)} = v$ equals $[x]_{s'(x:=v')} = v'$ in case e_ℓ is ℓ-visible (via (4)), and if not, this pair of traces can be neglected as respective schedule-fragments $\sigma = [\mathsf{in}\langle [e_\ell]_s, v\rangle]$ and $\sigma' = [\mathsf{in}\langle [e_\ell]_{s'}, v'\rangle]$ from the small-step semantics are not input equivalent (cf. Definition 1).

In comparison, there are two rules for the output command, one for a secure output, OutOk, and one for an insecure output, OutInsec shown in (2). If one wants to prove for a given case study that the insecure outcome $e \not: e_\ell$ is unreachable, one can check the result and presumption wrt. a frame assertion P that captures the path condition of the context in which the output was made, so that if $P * e \not: e_\ell$ is unsatisfiable the result is demonstrated to be unreachable.

Moreover, there are rules that expose branching on secrets as the test of **if** and **while** statements, and rule SeqInsec propagates an insecurity in the first part of a sequential composition similarly to an error.

5 Symbolic Execution

InsecSL's careful design, as a relational logic that resembles the non-relational ISL, means that its application can be automated via bi-abduction [5] based symbolic execution method for automatically deriving InsecSL judgements.

We formalise the symbolic execution method for ISL, atop InsecSL, proving that it yields a sound analysis method for automatically inferring InsecSL judgements. Ours is the first such symbolic execution method, for an under-approximate logic, to enjoy a mechanised proof of soundness.

To define our symbolic execution, it helps to introduce an extra program command **assume**(e). This command is not a "real" command in the sense that it cannot appear in program text. Instead, it is used to remember, during symbolic execution, which conditional branches have been followed along the current execution path. As we will see, our symbolic execution maintains a *trace* that records the execution path followed so far, in which assume commands **assume**(e) can appear. Their semantics is to evaluate the condition e and, if e holds to act as a no-op but otherwise execution gets stuck.

Our symbolic execution method stores the path followed so far. Doing so allows it to provide detailed information to the user when a vulnerability is detected (e.g. to tell precisely along which path the vulnerability arises). Doing so is also necessary to prove the soundness of our method, as explained later. The current path is stored as a *trace*, which is a list of pairs (c, P) where c

is a program command and P an InsecSL assertion. For convenience, traces are stored in *reverse* order. Each element (c, P) is understood to mean that command c was executed from symbolic state P, i.e. P represents the state before c was executed. We write the empty trace $[]$ (which represents that there has been no preceding symbolic execution), and the trace whose head is x and whose tail is xs as $x : xs$.

When a new spatial assertion F is inferred to make forward progress in symbolic execution, it is then *back-propagated* along the trace tr, causing F to be added into each of the assertions P in each element (c, P) of F. Given an assertion F, back-propagating it over trace tr produces the transformed trace tr', and operates in the expected way by successively appealing to the Frame rule. We define the procedure $\mathsf{backprop}_\ell(F, tr, tr')$ for doing this.

Definition 3 (Backprop). *For any assertion F, any security level ℓ, and any traces tr and tr' where each of them is a list of command-assertion pairs, $\mathsf{backprop}_\ell(F, tr, tr')$ holds if and only if: $tr = tr' = [] \vee (\exists c\ P\ F\ F'\ tr_2\ tr_2'.\ tr = (c, P) : tr' \wedge tr' = (c, P * F) : tr_2' \wedge mod(c) \cap fv(F) = \emptyset \wedge \mathsf{backprop}_\ell(F', tr_2, tr_2'))$*

Symbolic execution is then defined as follows. We define a judgement $\mathsf{symex}_\ell(tr,\ JQ,\ c,\ tr',\ JQ')$. Here c is a command, tr and tr' are traces, while JQ and JQ' are judgement *post assertions*, i.e. have one of the following forms each for some assertion Q: $ok : Q$, $err : Q$, or $insec : Q$. Trace tr and JQ represent the current state of symbolic execution before command c is executed, in the sense that tr is the trace followed up to this point and JQ represents the symbolic state immediately before c is executed. Executing c necessarily extends the trace (possibly also transforming it via back-propagation), yielding an updated trace tr' and a new post assertion JQ'.

The symbolic execution rules are shown in Fig. 3. When encountering branching, symbolic execution will flag insecurity (SeIfInsec) if the branch condition b is secret ($b = \mathbf{true} \not: \ell$); however it can also proceed (e.g. SeIfTrue) by assuming the branch condition (implicitly assuming it is non-secret). The rule SeOutInsec detects insecure outputs. Rules for inferring spatial predicates via bi-abduction follow their counterparts in ISL [12].

Theorem 2 (Soundness of Symbolic Execution). *For all commands c, security levels ℓ, post-assertions JQ and JQ' and all traces tr, produced by symbolic execution, i.e., $\mathsf{symex}_\ell([],\ JQ,\ c,\ tr,\ JQ')$ holds, we have tr is not empty. Furthermore, letting (c, P) denote the last element of tr, we have $\vdash_\ell [P]\ c\ [JQ']$.*

As mentioned earlier, the trace tr is not merely a user convenience but a necessary ingredient to prove soundness of the structural rules, like SeIfTrue above. Soundness of this rule for instance requires deducing a judgement $\vdash_\ell [P]\ c_0; c'\ [\epsilon :\ Q]$ given premise $\vdash_\ell [P]\ c_0; c\ [\epsilon :\ Q]$ and inductive hypothesis $\forall P\ Q.\ \vdash_\ell [P]\ c\ [\epsilon :\ Q] \implies \vdash_\ell [P]\ c'\ [\epsilon :\ Q]$. Unfortunately the premise is not strong enough to deduce some intermediate assertion R for which $\vdash_\ell [P]\ c_0\ [\epsilon :\ R]$ and $\vdash_\ell [R]\ c\ [\epsilon :\ Q]$ as required to instantiate the inductive hypothesis. Inclusion of trace tr allows us to express the necessary strengthening of the theorem.

$$\frac{}{\mathsf{symex}_\ell(tr,\ [ok\colon P],\ \mathbf{skip},\ (\mathbf{skip}, P) : tr,\ [ok\colon P])}\text{SESkip}$$

$$\frac{}{\mathsf{symex}_\ell(tr,\ [ok\colon P],\ \mathbf{assume}(b),\ (\mathbf{assume}(b), P) : tr,\ [ok\colon P * b])}\text{SEAsm}$$

$$\frac{}{\mathsf{symex}_\ell(tr,\ [ok\colon P],\ \mathbf{output}(el, e),\ (\mathbf{output}(el, e), P) : tr,\ [ok\colon P * e :: el])}\text{SEOut}$$

$$\frac{c = (\mathbf{output}(el, e))}{\mathsf{symex}_\ell(tr,\ [ok\colon P],\ L\colon c,\ (L\colon c, P) : tr,\ [insec(L)\colon P * e \not: el])}\text{SEOutInsec}$$

$$\frac{x' \notin fv(P)}{\mathsf{symex}_\ell(tr,\ [ok\colon P],\ x := e,\ (x := e, P) : tr,\ [ok\colon P[x'/x] * x = e[x'/x]])}\text{SEAssign}$$

$$\frac{x' \notin fv(P)}{\mathsf{symex}_\ell(tr,\ [ok\colon P],\ x := \mathbf{input}(e),\ (x := \mathbf{input}(e), P) : tr,\ [ok\colon P[x'/x] * x :: e[x'/x]])}\text{SEInput}$$

$$\frac{x' \notin fv(P)}{\mathsf{symex}_\ell(tr,\ [ok\colon P],\ x := \mathbf{alloc}(e),\ (x := \mathbf{alloc}(e), P) : tr,\ [ok\colon P[x'/x] * x \mapsto e[x'/x]])}\text{SEAlloc}$$

$$\frac{\mathsf{backprop}_\ell(M, tr, tr') \qquad x' \notin fv(Frame) \qquad p \mapsto e * Frame \overset{\ell}{\Longrightarrow} P * M}{\mathsf{symex}_\ell(tr,\ [ok\colon P],\ x := [p],\ (x := [p], P * M) : tr',\ [ok\colon x = e[x'/x] * (p \mapsto e * Frame)[x'/x]])}\text{SELoad}$$

$$\frac{\mathsf{backprop}_\ell(M, tr, tr') \qquad p \not\mapsto * Frame \overset{\ell}{\Longrightarrow} P * M}{\mathsf{symex}_\ell(tr,\ [ok\colon P],\ L\colon x := [p],\ (L\colon x := [p], P * M) : tr',\ [err(L)\colon p \not\mapsto * Frame])}\text{SELoadErr}$$

$$\frac{\mathsf{backprop}_\ell(M, tr, tr') \qquad p \mapsto e * Frame \overset{\ell}{\Longrightarrow} P * M}{\mathsf{symex}_\ell(tr,\ [ok\colon P],\ [p] := e',\ ([p] := e', P * M) : tr',\ [ok\colon p \mapsto e' * Frame])}\text{SEStore}$$

$$\frac{\mathsf{backprop}_\ell(M, tr, tr') \qquad p \not\mapsto * Frame \overset{\ell}{\Longrightarrow} P * M}{\mathsf{symex}_\ell(tr,\ [ok\colon P],\ L\colon [p] := e',\ (L\colon [p] := e', P * M) : tr',\ [err(L)\colon p \not\mapsto * Frame])}\text{SEStoreErr}$$

$$\frac{\mathsf{backprop}_\ell(M, tr, tr') \qquad p \mapsto e * Frame \overset{\ell}{\Longrightarrow} P * M}{\mathsf{symex}_\ell(tr,\ [ok\colon P],\ \mathbf{free}(p),\ (\mathbf{free}(p), P * M) : tr',\ [ok\colon p \not\mapsto * Frame])}\text{SEFree}$$

$$\frac{\mathsf{backprop}_\ell(M, tr, tr') \qquad p \not\mapsto * Frame \overset{\ell}{\Longrightarrow} P * M}{\mathsf{symex}_\ell(tr,\ [ok\colon P],\ L\colon \mathbf{free}(p),\ (L\colon \mathbf{free}(p), P * M) : tr',\ [err(L)\colon p \not\mapsto * Frame])}\text{SEFreeErr}$$

$$\frac{c = (\mathbf{if}\ b\ \mathbf{then}\ c1\ \mathbf{else}\ c2\ \mathbf{endif}) \qquad c1 \neq c2}{\mathsf{symex}_\ell(tr,\ [ok\colon P],\ L\colon c,\ (L\colon c, P * b = \mathbf{true} \not: \ell) : tr,\ [insec(L)\colon P * b = \mathbf{true} \not: \ell])}\text{SEIfInsec}$$

$$\frac{\mathsf{symex}_\ell(tr,\ [ok\colon P],\ \mathbf{assume}(b); c_1,\ tr',\ Q)}{\mathsf{symex}_\ell(tr,\ [ok\colon P],\ \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{endif},\ tr',\ Q)}\text{SEIfTrue}$$

Fig. 3. Symbolic execution rules.

This construction was not necessary for the pen-and-paper soundness proof of ISL [16,12] because for any single state there exists an ISL assertion that precisely describes that state, and hence the existence of the intermediate assertion R is trivial in ISL. The same is not true for InsecSL because InsecSL's assertions, while resembling unary ones, are evaluated relationally (cf. Section 4).

Our symbolic execution as described can be applied to the body of a function to infer InsecSL judgements that describe its internal behaviour. Such judgements must be transformed into summaries that describe the function's external behaviour. To do so we follow the same approach as in ISL [12]. For instance, consider the trivial function `void func(int x){ x = x + 1; }` that uselessly increments its argument `x`. Its internal behaviour is captured by the judgement $\vdash_\ell [\mathtt{x} = v]$ `x = x + 1` $[ok\colon v' = v * \mathtt{x} = v' + 1]$, where the logical variable v captures the initial value of `x`. Transforming this internal judgement into an external summary (after simplification) yields the summary $\vdash_\ell [\mathbf{emp}]$ `func(x)` $[ok\colon \mathbf{emp}]$.

6 Implementation

We implemented the symbolic execution procedure for automating the application of InsecSL in two tools: Underflow and Pulse-InsecSL. Underflow implements the entirety of InsecSL via *contextual, top-down* inter-procedural symbolic execution. Pulse-InsecSL on the other hand is a modification of the existing non-contextual, *bottom-up* inter-procedural symbolic execution method for ISL that is implemented in the Pulse-ISL plugin for Infer [12], which we modify to implement a useful subset of the InsecSL logic.

Underflow is a proof-of-concept tool, which we built by modifying an existing verifier for the over-approximate security separation logic SecCSL [10]. Underflow implements a top-down inter-procedural analysis in which individual functions (procedures) are analysed using the symbolic execution method of Section 5 to derive summaries for their behaviours.

When analysing a function $f()$ that calls another $g()$ Underflow attempts to apply all summaries known about $g()$. If none of them are applicable (i.e. applying them yields an inconsistent state), Underflow performs a contextual analysis of $g()$ to compute new summaries applicable at this callsite. To perform a contextual analysis of callee $g()$ from caller $f()$ we take the current symbolic state R and filter it to produce a state R' that describes only those parts of R relevant to the call. Underflow's present implementation does so using a fixed-point computation that identifies all pure formulae from R that mention arguments passed to $g()$ and values (transitively) related to those arguments by such pure formulae. It identifies all spatial assertions in R that describe parts of the heap reachable from those values, filtering everything else as irrelevant.

In contrast to Infer [16,12], Underflow does not unroll loops to a fixed bound. Instead it controls symbolic execution using two mechanisms. Firstly, for each program point it counts the number of paths that have so far passed through that point during analysis. When that number exceeds a configurable bound, additional paths are discarded. Additionally it monitors the latency of symbolically executing each program statement. When this latency gets too high (exceeds a configurable timeout), the current path is discarded. The former bound is reached only when unfolding relatively tight loops, while the latter attempts to maintain reasonable symbolic execution throughput. When analysing a function Underflow will avoid generating multiple summaries that report the same problem for

a single program point. UNDERFLOW reports *unconditional* (aka *manifest* [12]) bugs whose presumptions are **true**.

UNDERFLOW encodes all non-spatial formulae to SMT via a relational encoding which directly encodes their relational semantics (Fig. 5). Doing so necessarily duplicates each variable, meaning that SMT encodings of formulae are often relatively large. While this can impede scalability, it ensures that UNDERFLOW encodes the entirety of INSECSL in a semantically complete way.

PULSE-INSECSL takes a different design to UNDERFLOW, and makes maximum advantage of the fact that INSECSL is purposefully designed to be very similar to ISL [16], allowing its symbolic execution procedure (Section 5) to very closely resemble that for ISL also [12].

PULSE-INSECSL implements a non-trivial fragment of INSECSL. In this fragment, there are only two security levels ℓ: **low** (bottom) and **high** (top). The level of the attacker is **low**. Insecurity assertions $b \not: \mathbf{low}$ appear only over boolean expressions b and mention only the security level **low**. Security assertions $e :: \mathbf{low}$ do not appear directly. Instead, whenever an expression e is to be treated as **low** ($e :: \mathbf{low}$), the expression e is concretised, i.e. replaced by a concrete value (a constant). We refer to this process as *low concretisation*. Since constants are **low** by definition, concretising **low** expressions e ensures that PULSE-INSECSL treats them as **low** without having to perform a relational encoding of the security assertion $e :: \mathbf{low}$. In our current implementation, constants for concretisation are not chosen randomly, ensuring determinism.

Likewise, PULSE-INSECSL avoids having to perform relational encoding of insecurity assertions $b \not: \mathbf{low}$ by soundly encoding them as follows. In particular:

$$\mathbf{sat}\ b \not: \mathbf{low} \iff \mathbf{sat}\ b \text{ and } \mathbf{sat}\ \neg b.$$

Thus satisfiability of insecurity assertions over boolean conditions b can be checked via unary (non-relational) satisfiability checking.

With these two techniques, PULSE-INSECSL automates INSECSL reasoning directly within the existing symbolic execution framework for ISL with minimal modifications, inheriting Infer's highly optimised implementation and scalability. In this implementation, PULSE-INSECSL performs symbolic execution in a bottom-up fashion: each function is analysed in isolation from all others to produce summaries. Loops are unrolled up to a fixed bound, making symbolic execution entirely deterministic.

7 Evaluation

We evaluate both UNDERFLOW and PULSE-INSECSL on the programs, listed in Table 1. The `auction` sample is the synthetic auction case study from Fig. 1. The samples `ctselect`, `ctsort`, `haclpolicies`, `kremlib`, `libsodiumutils`, `opensslutil`, `ssl3cbcrem`, `tls1lucky13`, `tls1patched` are cryptographic library code, drawn from benchmarks for the Binsec/Rel tool [7]. Samples `ctselect`, `ctsort` and `tls1lucky13` contain known vulnerabilities. Most are libraries of basic helper

Table 1. Tool evaluation results. For each sample we record its size in (SLOC) and the number of *top-level* functions analysed (# funs). The third column (sec?) indicates whether the sample had no security vulnerabilities known a-priori. Analysis time of UNDERFLOW for each sample is reported in seconds. PULSE-INSECSL analysed each sample in less than one second. The unique top-level bugs reported we break down into memory safety errors (# err, for UNDERFLOW) and information leaks (# insec, for both). (†) indicates samples that were analysed in PULSE-INSECSL with a manually set loop unrolling bound. (‡) indicates samples that were analysed in UNDERFLOW with an increased symbolic execution pruning and SMT timeout of 600 seconds.

				UNDERFLOW			PULSE-INSECSL
Sample	SLOC	# funs	sec?	time (s)	# err	# insec	# insec
`auction` (†)	172	1	✗	195	0	1	1
`ctselect`	27	5	✗	1	0	1	1
`ctsort`	57	3	✗	5	0	2	**7**
`cttkinner`	77	3	✓	5	0	0	0
`haclpolicies`	34	1	✓	50	0	0	0
`hex`	178	2	✓	80	0	1	1
`int31` (‡)	1923	60	✓	708	1	2	-
`kremlib`	68	10	✓	2	0	0	0
`libsodiumutils`	115	3	✓	380	0	1	1
`opensslutil`	84	7	✓	1	0	0	0
`oram1`	167	4	✓	27	0	1	1
`ssl3cbcrem`	111	1	✓	10	0	0	0
`tls1lucky13`	122	1	✗	119	1	4	**6**
`tls1patched`	229	1	✓	192	2	2	0

routines, except for `ssl3cbcrem`, `tls1lucky13` and `tls1patched`. The latter two are the vulnerable and patched versions of the infamous "Lucky13" TLS vulnerability [1]. The remaining samples are drawn from the Constant-Time Toolkit (CTTK) (https://github.com/pornin/CTTK): `cttkinner` is a library of basic helper functions, `hex` is purportedly constant-time routines for converting to/from binary and hexadecimal strings; `int31` is drawn from big integer library; `oram1` is a basic oblivious RAM (ORAM) library.

Accuracy and Bug Discovery. For the known vulnerable samples, UNDERFLOW and PULSE-INSECSL correctly detect the known vulnerabilities. UNDERFLOW additionally identifies an out-of-bounds array access in the big integer library `int31`. This vulnerability we confirmed by fuzzing the affected code with libFuzzer and AddressSanitizer enabled, and was subsequently confirmed by the developer of the CTTK library. UNDERFLOW also identified an undocumented information leak in the `hex` CTTK sample, which leaks the location of non-hex characters in strings. Upon reporting this issue to the developer, we were informed it was intended behaviour. This behaviour was also detected by PULSE-INSECSL. UNDERFLOW identified two information leaks also in the `int31` library in routines for copying one big integer to another. In particular, if the destina-

tion big integer is not initialised, then these routines can leak information about the destination memory contents. Limitations in Pulse-InsecSL's current implementation prevent it from running on `int31` at the time of writing.

The information leak identified by Underflow in `libsodiumutils` is similar to that in `hex` and occurs in a routine for converting hex strings to binary, leaking information if the hex string contains non-hex characters. Both tools correctly identify the "Lucky13" vulnerability in `tls1lucky13`. Underflow additionally identifies an out-of-bounds array access in this legacy (now patched) code, heretofore undiagnosed. The two information leaks that Underflow identifies in the patched "Lucky13" code `tls1patched` are due to if-conditions that branch on secrets but, which many compilers optimise away and hence why this sample is considered to have no known vulnerabilities. Thus whether one regards these reports as true or false positives depends on how the code is compiled.

In two samples, Pulse-InsecSL reports additional information leaks not reported by Underflow (bold entries). These arise because Pulse-InsecSL treats expressions like `(a > b) - 1` as if they branch on the boolean condition `a > b`. Indeed, `gcc` 13.1 will compile such code to a conditional jump when compiled at the lowest optimisation level `-O0` for x86-64, so we regard these reports as true positives; however we note that on all higher optimisation levels all modern C compilers will compile such expressions to straight line code that doesn't leak.

Performance. Pulse-InsecSL is orders of magnitude faster than Underflow, in general. In particular, while Underflow can take minutes to run on some samples, Pulse-InsecSL takes no more than a second to analyse each sample. This should be expected, for a number of reasons. Firstly, recall that Underflow uses a timeout mechanism to prune paths during symbolic execution in which paths are pruned when symbolic execution of individual statements becomes too slow. On the other hand Pulse-InsecSL uses a deterministic strategy to prune paths, by choosing to unroll loops up to a fixed bound only (by default, once). Thus programs with unbounded loops, like `auction`, take a long time for Underflow to analyse because it keeps unrolling the main loop until symbolic execution becomes sufficiently slow due to the growing size of the path condition. This also means that Underflow may explore loops many more times (and so uncover more behaviours) than Pulse-InsecSL in general, so the amount of symbolic execution that the former performs on a given program is often much greater than the second. To scale Underflow to the `int31` sample required increasing its default path pruning timeout. Thus we might expect that scaling Underflow beyond samples of this size may be challenging. Pulse-InsecSL on the other hand suffers no such scalability challenges.

Secondly, Underflow makes use of an external SMT solver in which all non-spatial assertions are given a relational (i.e. two-execution) encoding to SMT, with very little simplification before formulae are encoded to SMT. On the other hand, Pulse-InsecSL is designed to avoid the need for relational assertion encoding and in any case uses a highly performant in-built satisfiability checking library while continually performing aggressive formula simplification. Pulse-InsecSL benefits from many years of development effort and optimisation, while

having a much simpler problem to solve (unary symbolic execution). UNDERFLOW on the other hand has far fewer optimisations and has not been designed for speed, while solving a much harder problem (relational symbolic execution).

We note that the analysis times of PULSE-INSECSL also dwarf the reported analysis times of the relational symbolic executor Binsec/Rel [7] which, like UNDERFLOW, takes minutes to analyse some samples (e.g. the "Lucky13" sample for which it requires over an hour of execution time [7, Table III]).

8 Related Work and Conclusion

Our logic INSECSL is the relational analogue of ISL [16], in the same way that Security Concurrent Separation Logic (SECCSL) [10] is the relational analogue of traditional separation logic [17,14]. INSECSL can also be seen as the under-approximate dual of SECCSL, in the same way that Incorrectness Logic [15] is the under-approximate dual of Hoare logic. Despite INSECSL being relational, our symbolic execution procedure is purposefully essentially identical to that for ISL [16,12]. This allowed us to implement it as an extension of the existing symbolic execution implementation for ISL in the Infer tool.

Our symbolic execution procedure is also somewhat similar to relational symbolic execution [11] (RSE). However, RSE is not defined for programs with nondeterminism (including from dynamic memory allocation or external input, both of which we support). Indeed, RSE was proved sound with respect to over-approximate Relational Hoare logic [4], whereas ours is based on our under-approximate logic INSECSL. We conjecture that extending RSE to handle nondeterminism would be non-trivial, not least because over-approximate logics cannot precisely describe errors in nondeterministic programs (as we noted in Section 1). Unlike RSE, which is a whole-program analysis, our method is compositional, allowing it also be applied incrementally.

The recently developed Outcome Logic [20] unifies underapproximative and overapproximative reasoning within a uniform framework. It would be interesting to instantiate this approach with our relational setting.

Declassification is the act of intentionally revealing sensitive information in a controlled way. This aspect is orthogonal to the contribution of INSECSL and could be incorporated with standard approaches [2].

We have presented INSECSL, a logic that soundly discovers insecurities in program code. The logic strikes a particular balance: Despite being based on a relational semantic foundation, it is fairly straight-forward to automate and inherits many strengths of comparable approaches like ISL, foremost being compositional. We have demonstrated that it is capable of precise reasoning about real insecurities (and errors) in C source code.

References

1. Al Fardan, N.J., Paterson, K.G.: Lucky thirteen: Breaking the tls and dtls record protocols. In: IEEE Symposium on Security and Privacy. pp. 526–540. IEEE (2013)

2. Banerjee, A., Naumann, D.A., Rosenberg, S.: Expressive declassification policies and modular static enforcement. In: IEEE Symposium on Security and Privacy. pp. 339–353. IEEE (2008)
3. Barthe, G., Blazy, S., Grégoire, B., Hutin, R., Laporte, V., Pichardie, D., Trieu, A.: Formal verification of a constant-time preserving c compiler. PACMPL **4**(POPL), 1–30 (2020)
4. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: POPL. pp. 14–25 (2004)
5. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL. pp. 289–300 (2009)
6. Clarkson, M.R., Schneider, F.B.: Hyperproperties. Journal of Computer Security **18**(6), 1157–1210 (2010)
7. Daniel, L.A., Bardin, S., Rezk, T.: BINSEC/REL: Efficient relational symbolic execution for constant-time at binary-level. In: IEEE Symposium on Security and Privacy. pp. 1021–1038. IEEE (2020)
8. De Vries, E., Koutavas, V.: Reverse hoare logic. In: SEFM. pp. 155–171 (2011)
9. Eilers, M., Müller, P., Hitz, S.: Modular product programs. In: ESOP. pp. 502–529 (2018)
10. Ernst, G., Murray, T.: SecCSL: Security concurrent separation logic. In: CAV. pp. 208–230 (2019)
11. Farina, G.P., Chong, S., Gaboardi, M.: Relational symbolic execution. In: PPDP. pp. 1–14 (2019)
12. Le, Q.L., Raad, A., Villard, J., Berdine, J., Dreyer, D., O'Hearn, P.W.: Finding real bugs in big programs with incorrectness logic. PACMPL **6**(OOPSLA1), 1–27 (2022)
13. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The program counter security model: Automatic detection and removal of control-flow side channel attacks. In: International Conference on Information Security and Cryptology. pp. 156–168. Springer (2005)
14. O'Hearn, P.W.: Resources, concurrency and local reasoning. In: CONCUR. pp. 49–67. Springer (2004)
15. O'Hearn, P.W.: Incorrectness logic. PACMPL **4**(POPL), 1–32 (2019)
16. 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: CAV (2020)
17. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74. IEEE (2002)
18. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1), 5–19 (2003)
19. Yang, H.: Relational separation logic. Theoretical Computer Science **375**(1-3), 308–334 (2007)
20. Zilberstein, N., Dreyer, D., Silva, A.: Outcome logic: A unifying foundation for correctness and incorrectness reasoning. PACMPL **7**(OOPSLA1), 522–550 (2023)

A Appendix

A.1 Language Semantics

The small-step semantics for the language over which InsecSL is defined is defined in Fig. 4.

$$\langle \textbf{run } \text{“}\textbf{skip}\text{”}\ s\ h\rangle \xrightarrow{[\tau]} \langle \textbf{stop}\ s\ h\rangle \qquad \langle \textbf{run } \text{“}x := \textbf{input}(e)\text{”}\ s\ h\rangle \xrightarrow{[\mathsf{in}\langle [e]_s, v\rangle]} \langle \textbf{stop}\ s(x := v)\ h\rangle$$

$$\langle \textbf{run } \text{“}x := e\text{”}\ s\ h\rangle \xrightarrow{[\tau]} \langle \textbf{stop}\ s(x := [e]_s)\ h\rangle \qquad \frac{a = [p]_s \qquad a \notin dom(h) \vee h(a) = \bot}{\langle \textbf{run } \text{“}x := [p]\text{”}\ s\ h\rangle \xrightarrow{[\tau]} \langle \textbf{abort}\ s\ h\rangle}$$

$$\frac{a = [p]_s \qquad h(a) = v}{\langle \textbf{run } \text{“}x := [p]\text{”}\ s\ h\rangle \xrightarrow{[\tau]} \langle \textbf{stop}\ s(x := v)\ h\rangle} \qquad \frac{a = [p]_s \qquad a \notin dom(h) \vee h(a) = \bot}{\langle \textbf{run } \text{“}[p] := e\text{”}\ s\ h\rangle \xrightarrow{[\tau]} \langle \textbf{abort}\ s\ h\rangle}$$

$$\frac{a = [p]_s \qquad h(a) = v}{\langle \textbf{run } \text{“}[p] := e\text{”}\ s\ h\rangle \xrightarrow{[\tau]} \langle \textbf{stop}\ s\ h(a := [e]_s)\rangle}$$

$$\frac{a \notin dom(h) \vee h(a) = \bot}{\langle \textbf{run } \text{“}x := \textbf{alloc}(e)\text{”}\ s\ h\rangle \xrightarrow{[\mathsf{allocate}\langle a\rangle]} \langle \textbf{stop}\ s(x := a)\ h(a := [e]_s)\rangle}$$

$$\frac{a = [p]_s \qquad h(a) = v}{\langle \textbf{run } \text{“}\textbf{free}(p)\text{”}\ s\ h\rangle \xrightarrow{[\tau]} \langle \textbf{stop}\ s\ h(a := \bot)\rangle} \qquad \frac{a = [p]_s \qquad a \notin dom(h) \vee h(a) = \bot}{\langle \textbf{run } \text{“}\textbf{free}(p)\text{”}\ s\ h\rangle \xrightarrow{[\tau]} \langle \textbf{abort}\ s\ h\rangle}$$

$$\langle \textbf{run } \text{“}\textbf{output}(e_\ell, e)\text{”}\ s\ h\rangle \xrightarrow{\mathsf{out}\langle [e_\ell]_s, [e]_s\rangle} \langle \textbf{stop}\ s\ h\rangle \qquad \frac{\langle \textbf{run } \text{“}c_1\text{”}\ s\ h\rangle \xrightarrow{\sigma} \langle \textbf{abort}\ s'\ h'\rangle}{\langle \textbf{run } \text{“}c_1; c_2\text{”}\ s\ h\rangle \xrightarrow{\sigma} \langle \textbf{abort}\ s'\ h'\rangle}$$

$$\frac{\langle \textbf{run } \text{“}c_1\text{”}\ s\ h\rangle \xrightarrow{\sigma} \langle \textbf{stop}\ s'\ h'\rangle}{\langle \textbf{run } \text{“}c_1; c_2\text{”}\ s\ h\rangle \xrightarrow{\sigma} \langle \textbf{run } \text{“}c_2\text{”}\ s'\ h'\rangle} \qquad \frac{\langle \textbf{run } \text{“}c_1\text{”}\ s\ h\rangle \xrightarrow{\sigma} \langle \textbf{run } \text{“}c_1'\text{”}\ s'\ h'\rangle}{\langle \textbf{run } \text{“}c_1; c_2\text{”}\ s\ h\rangle \xrightarrow{\sigma} \langle \textbf{run } \text{“}c_1'; c_2\text{”}\ s'\ h'\rangle}$$

$$\frac{[b]_s = \textbf{true}}{\langle \textbf{run } \text{“}\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{endif}\text{”}\ s\ h\rangle \xrightarrow{[\tau]} \langle \textbf{run } \text{“}c_1\text{”}\ s\ h\rangle}$$

$$\frac{[b]_s \neq \textbf{true}}{\langle \textbf{run } \text{“}\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{endif}\text{”}\ s\ h\rangle \xrightarrow{[\tau]} \langle \textbf{run } \text{“}c_2\text{”}\ s\ h\rangle}$$

$$\frac{[b]_s = \textbf{true}}{\langle \textbf{run } \text{“}\textbf{while}\ b\ \textbf{do}\ c\ \textbf{done}\text{”}\ s\ h\rangle \xrightarrow{[\tau]} \langle \textbf{run } \text{“}c; \textbf{while}\ b\ \textbf{do}\ c\ \textbf{done}\text{”}\ s\ h\rangle}$$

$$\frac{[b]_s \neq \textbf{true}}{\langle \textbf{run } \text{“}\textbf{while}\ b\ \textbf{do}\ c\ \textbf{done}\text{”}\ s\ h\rangle \xrightarrow{[\tau]} \langle \textbf{run } \text{“}\textbf{skip}\text{”}\ s\ h\rangle} \qquad \frac{[b]_s = \textbf{true}}{\langle \textbf{run } \text{“}\textbf{assume}(b)\text{”}\ s\ h\rangle \xrightarrow{[\tau]} \langle \textbf{stop}\ s\ h\rangle}$$

Fig. 4. Small step semantics of the language for InsecSL. For a function f we write $f(x := v)$ to denote function update, i.e. to abbreviate the function that behaves like f everywhere except for the argument x for which it returns v.

Using the abbreviations:

$$(s,h) \vDash e_p \mapsto e_v \iff h = \{[e_p]_s \mapsto [e_v]_s\}$$
$$(s,h) \vDash e_p \not\mapsto \iff h = \{[e_p]_s \mapsto \bot\}$$

$$\begin{aligned}
(s,h)\ (s',h') \vDash_\ell e &\iff [e]_s = \mathbf{true} \wedge [e]_{s'} = \mathbf{true} \wedge h = h' = \emptyset \\
(s,h)\ (s',h') \vDash_\ell e :: e_\ell &\iff [e_\ell]_s = [e_\ell]_{s'} \wedge ([e_\ell]_s \sqsubseteq \ell \implies [e]_s = [e]_{s'}) \wedge h = h' = \emptyset \\
(s,h)\ (s',h') \vDash_\ell e \not:: e_\ell &\iff [e_\ell]_s = [e_\ell]_{s'} \wedge [e_\ell]_s \sqsubseteq \ell \wedge [e]_s \neq [e]_{s'} \wedge h = h' = \emptyset \\
(s,h)\ (s',h') \vDash_\ell \mathbf{emp} &\iff h = h' = \emptyset \\
(s,h)\ (s',h') \vDash_\ell e_p \mapsto e_v &\iff (s,h) \vDash e_p \mapsto e_v \wedge (s',h') \vDash e_p \mapsto e_v \\
(s,h)\ (s',h') \vDash_\ell e_p \not\mapsto &\iff (s,h) \vDash e_p \not\mapsto \wedge\ (s',h') \vDash e_p \not\mapsto \\
(s,h)\ (s',h') \vDash_\ell P_1 * P_2 &\iff \text{there are disjoint subheaps } h_1, h_2, \text{ and } h_1', h_2' \\
&\qquad \text{where } h = h_1 \uplus h_2 \wedge h' = h_1' \uplus h_2' \\
&\qquad \text{such that } (s,h_1)\ (s',h_1') \vDash_\ell P_1 \text{ and } (s,h_2)\ (s',h_2') \vDash_\ell P_2 \\
(s,h)\ (s',h') \vDash_\ell \exists x.\ P\ x &\iff \text{there are values } v,\ v' \\
&\qquad \text{such that } (s(x := v), h)\ (s'(x := v'), h') \vDash_\ell P \\
(s,h)\ (s',h') \vDash_\ell P \implies Q &\iff (s,h)\ (s',h') \vDash_\ell P \text{ implies } (s,h)\ (s',h') \vDash_\ell Q \\
(s,h)\ (s',h') \vDash_\ell \mathbf{false} &\qquad \text{never}
\end{aligned}$$

Fig. 5. Semantics of InsecSL assertions.

A.2 Assertion Semantics

The semantics of InsecSL assertions are given in Fig. 5. Most of these are familiar and inherited from their counterparts in SecCSL [10]. As in SecCSL, InsecSL assertions are given a relational semantics [19], i.e. are evaluated against a pair of states $(s,h),\ (s',h')$. We write $(s,h)\ (s',h') \vDash_\ell P$ to mean that assertion P holds in the pair of states $(s,h)\ (s',h')$. The security level ℓ denotes the security level of the attacker (see Section 3).

Implication and **false** are lifted in the obvious way. $\exists\ x.\ P\ x$ holds when a pair of values v, v' can be found for x in the left and right states respectively to make P hold. Pure expressions e are given a boolean interpretation by testing whether they evaluate to a distinguished value **true** in both states. Similarly, spatial assertions like **emp**, $e_p \mapsto e_v$ and $e_p \not\mapsto$ essentially assert the standard separation logic assertion semantics over both states. Separating conjunction lifts its ordinary separation logic counterpart over pairs of states: $P_1 * P_2$ holds when each heap can be partitioned into a left and right part, so that P_1 holds of the two left parts, and P_2 does likewise for the two right parts.

The semantics of $e :: e_\ell$ remain unchanged from SecCSL, and assert that e is known to the attacker if the attacker is able to observe e_ℓ-level outputs or, equivalently, e is known to the attacker if the attacker's level is greater than or equal to that denoted by e_ℓ. Recall that ℓ denotes the attacker's security level. We say that in a pair of states the attacker knows the value of some expression e, if e evaluates to identical values in those two states. Thus $e :: e_\ell$ holds between

two states precisely when, if the level denoted by e_ℓ is observable to the attacker ($[e_\ell]_s \sqsubseteq \ell$), the two states agree on the value of e.

Agreement on e between the two states formalises that the attacker knows e. For this reason, *disagreement* on e formalises that the attacker has some uncertainty about e. Hence, the semantics for $e \not:\!\!: e_\ell$.

A.3 Extending InsecSL to Constant-Time Security

We noted earlier in Section 3 that the security property and attacker model targeted by InsecSL is weaker than that of *constant-time security* [3]. InsecSL forbids a program to explicitly output or branch on secrets. Constant-time security additionally forbids a program from performing secret-dependent memory accesses.

Extending InsecSL to constant-time security is straightforward. We briefly sketch how. Doing so adds additional rules for loading and storing to the heap to detect insecurity. Similarly to OutInsec, these rules have in their result that the pointer p being loaded from (respectively stored to) is not known to the attacker: $p \not:\!\!: \ell$. The existing OK rules have the converse added to their results: $p :: \ell$.

The semantics of the language (Appendix A.1) is extended to record in the schedule σ the address of each pointer that is loaded from and stored to, effectively making these outputs of the program. The security property then imposes the extra requirement that in the two executions, these addresses are identical.

Soundness then follows from a similar argument as that for the existing output rules.