SecCSL: Security Concurrent Separation Logic

Gidon Ernst[1] and Toby Murray[2]

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

Abstract. We present SecCSL, a concurrent separation logic for proving expressive, data-dependent information flow security properties of low-level programs. SecCSL is considerably more expressive, while being simpler, than recent compositional information flow logics that cannot reason about pointers, arrays etc. To capture security concerns, SecCSL adopts a relational semantics for its assertions. At the same time it inherits the structure of traditional concurrent separation logics; thus SecCSL reasoning can be automated via symbolic execution. We demonstrate this by implementing SecC, an automatic verifier for a subset of the C programming language, which we apply to a range of benchmarks.

1 Introduction

Software verification successes abound, whether via interactive proof or via automatic program verifiers. While the former has yielded individual, deeply verified software artifacts [24,21,25] primarily by *researchers*, the latter appears to be having a growing impact on *industrial* software engineering [36,11,39].

At the same time, recent work has heralded major advancements in program logics for reasoning about secure *information flow* [34,23,33]—i.e. whether programs properly protect their secrets—yielding the first general program logics and proofs of information flow security for non-trivial concurrent programs [34]. Yet so far, such logics have remained confined to interactive proof assistants, making them practically inaccessible to industrial developers.

This is not especially surprising. The Covern logic [34], for example, pays for its generality with regard to expressive security policies, in terms of complexity. Worse, these logics reason only over very simple toy programming languages, which even lack support for pointers, arrays, and structures, Their complexity, we argue, hinders proof automation and makes scaling up these logics to real-world languages impractical. How, therefore, can we leverage the power of existing automatic deductive verification approaches for security proofs?

In this paper we present *Security Concurrent Separation Logic* (SecCSL), which achieves an unprecedented combination of simplicity, power, and ease of automation by capturing core concepts such as data-dependent variable sensitivity [50,27,31], and shared invariants on sensitive memory [34] in the familiar style of Concurrent Separation Logic (CSL) [38], as exemplified in Section 2.

Prior work [14,20] has noted the promise of separation logic for reasoning about information flow yet, to date, that promise remains unrealised. Indeed,

the only two prior encodings of information flow concepts into separation logics which we are aware of have overlooked crucial features like concurrency [14], and lack the ability to separately specify the sensitivity of *values* and memory *locations* as we explain in Section 2. The logic in [20] lacks soundness arguments altogether while [14] fail to satisfy basic properties needed for automation (see the discussion following Proposition 1).

Designing a logic with the right combination of features, with the right semantics, is therefore non-trivial. To manage this, SecCSL assertions have a *relational* interpretation [6,49] over a standard heap model (Section 3). This allows one to canonically encode information flow concepts while maintaining the approach and structure of traditional CSL proofs. We have mechanised the soundness of SecCSL in Isabelle/HOL [37]. To do so we adapted existing proof techniques for the soundness of CSL [46] into a compositional information flow security property (Section 4) that, like SecCSL itself, is simple and powerful.

To demonstrate SecCSL's ease of use and capacity for automation, we implemented the prototype tool SecC (Section 5). We target C because it dominates low-level security-critical code. SecC automates SecCSL reasoning via symbolic execution, in the style of contemporary Separation Logic program verifiers like VeriFast [22], Viper [30], and Infer [10]. SecC correctly analyzes well-known benchmark problems (collected in [17]) within a few milliseconds; and we verify a variant of the CDDC case study [5] from the Covern project. Our Isabelle theories, the open source prototype tool SecC, and examples are available online at `https://covern.org/secc` [18].

2 An Overview of SecCSL

2.1 Specifying Information Flow Control in SecCSL

Consider the program in Fig. 1. It maintains a global pointer `rec` to a shared record, protected by the lock `mutex`. The `is_classified` field of the record identifies the confidentiality of the record's `data`: when `is_classified` is true, the value stored in the `data` field is confidential, and otherwise it is safe to release publicly. The left thread outputs the data in the record whenever it is public by writing to the (memory mapped) output device register pointer `OUTPUT_REG` (here also protected by `mutex`). The right thread updates the record, ensuring its contents is not confidential, here by clearing its `data`.

Suppose assigning a value d to the `OUTPUT_REG` register causes d to be outputted to a publicly-visible location. Reasoning, then, that the example is secure requires capturing that (1) the `data` field of the record pointed to by `rec` is confidential precisely when the record's `is_classified` field says it is, and (2) data sink `OUTPUT_REG` should never have confidential data written to it. Therefore the example only ever writes non-confidential data into `OUTPUT_REG`.

Condition (1) specifies the sensitivity of a data *value* in memory, whereas condition (2) specifies the sensitivity of the data that a memory *location* (i.e. data sink) is permitted to hold. Prior security separation logics [14,20] reason only

```
/* globals shared between the two threads */
struct record { bool is_classified; int data; };
struct record * rec = /* ... initialisation omitted ... */;
volatile int * const OUTPUT_REG = /* memory-mapped IO device register */;
```

```
/* thread 1: output the record */
while(true) {
    lock(mutex);
    if (!rec->is_classified)
        *OUTPUT_REG = rec->data;
    unlock(mutex); }
```

```
/* thread 2: edit the record */
lock(mutex);
/* clear the record */
rec->is_classified = FALSE;
rec->data = 0;
unlock(mutex);
```

Fig. 1. Example of Concurrent Information Flow.

about value-sensitivity condition (1) but, as we explain below, both are needed. Like those prior logics, in SecCSL one specifies the sensitivity of the value denoted by an expression e via a security *label* ℓ: the assertion $e :: \ell$ means that the sensitivity of the value denoted by expression e is at most ℓ. Security labels are drawn from a lattice with top element `high` (denoting the most confidential information), bottom element `low` (denoting public information), and ordered via $\sqsubseteq$: $\ell \sqsubseteq \ell'$ means that information labelled with ℓ' is at least as sensitive as that labelled by ℓ. Using this style of assertion, in conjunction with standard separation logic connectives (explained below), condition (1) can be specified as:

$$\exists c\ d.\ \texttt{rec} \mapsto (c, d) \land c :: \texttt{low} \land d :: (c\ ?\ \texttt{high} : \texttt{low}) \tag{1}$$

The $\mapsto$ is the standard separation logic points-to connective: $e \mapsto e'$ means the memory location denoted by expression e holds the value denoted by e'. Thus (1) can be read as saying that the `rec` pointer points to a pair of values (c, d). The first c (the value of the `is_classified` field) is public. The sensitivity of the second d (the value of the `data` field) is given by the value of the first c: it is `high` when c is true and is `low` otherwise. SecCSL integrates such reasoning about *value-dependent* sensitivity [50,27,31] neatly with functional properties of low-level data structures, which we think is more natural and straightforward than the approach of [35,34] that keeps the two concerns separate.

Value-sensitivity assertion $e :: \ell$ is a judgement on the maximum sensitivity of the data *source(s)* from which e has been derived. Location-sensitivity assertions, on the other hand, are used to specify security policies on data *sinks* like `OUTPUT_REG`. These assertions augment the separation logic points-to predicate with a security label ℓ, and are used to specify which parts of the memory are observable to the attacker (and so must never contain sensitive information): $e \overset{\ell}{\mapsto} e'$ means that the value denoted by the expression e' is present in memory at the location denoted by e, and additionally that at all times the sensitivity of the value stored in that locations is never allowed to exceed ℓ. Thus in SecCSL, $e \mapsto e'$ abbreviates $e \overset{\texttt{high}}{\longmapsto} e'$. In Fig. 1, that `OUTPUT_REG` is publicly-observable can be specified as: $\exists v.\ \texttt{OUTPUT\_REG} \overset{\texttt{low}}{\longmapsto} v$ (2)

2.2 Reasoning in SecCSL

SecCSL judgements have the form: $\ell_A \vdash \{P\}\ c\ \{Q\}$ (3)

Here ℓ_A is the *attacker security level*, c is the (concurrent) program command being executed, and P and Q are the program's pre- resp. postcondition. Judgement (3) means that if the program c begins in a state satisfying its precondition P then, when it terminates, the final state will satisfy its postcondition Q. Analogously to [44] the program is guaranteed to be memory safe. We defer a description of ℓ_A and the implied security property to Section 2.3.

As with traditional CSLs, SecCSL is geared towards reasoning over shared-memory programs that use lock-based synchronisation. Each lock l has an associated invariant $\mathrm{inv}(l)$, which is simply a predicate, like P or Q in (3), that describes the shared memory that the lock protects. In Fig. 1, where the lock `mutex` protects the shared pointer `rec` and `OUTPUT_REG`, the associated invariant $\mathrm{inv}(\texttt{mutex})$ is simply the conjunction of (1) and (2).

$$(\exists c\ d.\ \texttt{rec} \mapsto (c, d) \land c :: \texttt{low} \land d :: (c\ ?\ \texttt{high} : \texttt{low})) \star (\exists v.\ \texttt{OUTPUT\_REG} \overset{\texttt{low}}{\longmapsto} v) \quad (4)$$

Separating conjunction $P \star Q$ asserts that the assertions P and Q both hold and, additionally, that the memory locations referenced by P and Q respectively do not overlap. Thus SecCSL invariants, like SecCSL assertions, describe together both functional properties (e.g. `rec` is a valid pointer) and security concerns (e.g. the `OUTPUT_REG` location is publicly visible) of the shared state.

When acquiring a lock one gets to assume that the lock's invariant holds [38]. Subsequently, when releasing the lock one must prove that the invariant has been re-established. For example, when reasoning about the code of the left-thread in Fig. 1, upon acquiring the `mutex`, SecCSL adds formula (4) to the intermediate assertion, which allows proving that the loop body is secure. When reasoning about the right thread, one must prove that the invariant has been re-established when it releases the mutex. This is the reason e.g. that the right thread must clear the `data` field after setting `is_classified` to false.

Reasoning in SecCSL proceeds forward over the program text according to the rules in Fig. 4 on page 10. When execution forks, as in Fig. 1, one reasons over each thread individually. For Fig. 1, SecCSL requires proving that the guard of the `if`-condition is `low`, i.e. that the program is not branching on a secret (rule If in Fig. 4), which would correspond to a timing channel—see Section 2.3 below. This follows from the part $c :: \texttt{low}$ of invariant (4). Secondly, after the write to `OUTPUT_REG`, SecCSL requires that the expression that is being written to the location `OUTPUT_REG` has sensitivity `low` (rule Store in Fig. 4). This follows from $d :: (c\ ?\ \texttt{high} : \texttt{low}))$ in the invariant, which simplifies to $d :: \texttt{high}$ given the guard $c \equiv \texttt{true}$ of the `if`-statement. Finally, when the right thread releases `mutex`, invariant (4) holds for the updated contents of `rec` (rule Unlock in Fig. 4).

2.3 Security Intuition and Informal Security Property

But what does security mean in SecCSL? Indeed, the SecCSL a judgement $\ell_A \vdash \{P\}\ c\ \{Q\}$ additionally implies that the program c does not leak any sensitive information during its execution to potential attackers.

The attacker security level ℓ_A in (3) represents an upper bound on the parts of the program's memory that a potential, passive attacker is assumed to be able to observe before, during, and after the program's execution. Intuitively this encompasses all memory locations whose sensitivity is $\sqsubseteq \ell_A$. Which memory locations have sensitivity $\sqsubseteq \ell_A$ is defined by the *location-sensitivity* assertions in the precondition P and the lock invariants: A memory location *loc* is visible to the ℓ_A attacker iff P or a lock invariant contains some $e \overset{e_l}{\longmapsto} e'$ and in the program's initial state e evaluates to *loc* and e_l evaluates to some label ℓ such that $\ell \sqsubseteq \ell_A$ (see Fig. 3 on page 7).

Which data is sensitive and should not be leaked to the ℓ_A attacker is defined by the *value-sensitivity* assertions in P and the lock invariants: an expression e is sensitive when P or a lock invariant contains some $e :: e_l$ and in the program's initial state e_l evaluates to some ℓ with $\ell \not\sqsubseteq \ell_A$. Security, then, requires that in all intermediate states of the program's execution no sensitive data (as defined by value-sensitivity assertions) can be inferred via the attacker-observable memory (as defined by location-sensitivity assertions).

SecCSL proves a *compositional* security property that formalises this intuition (see Definition 3 on page 12). Since the property needs to be compositional with regards to concurrent execution, the resulting security property is *timing sensitive*, meaning that not only must the program never reveal sensitive data into attacker-observable memory locations but the times at which it updates these memory locations cannot depend on sensitive data. It is well-known that timing-insensitive security properties are not compositional under standard scheduling models [48,34]. For this reason SecCSL forbids programs from branching on sensitive values. We believe that this restriction could in principle be relaxed in the future via established techniques [29,28].

SecCSL's top-level soundness (Section 4) formalises the above intuitive definition of security in the style of traditional *noninterference* [19] that compares two program executions with respect to the observations that can be made by an attacker. SecCSL adopts a *relational* interpretation for the assertions P and Q, and the lock invariants, in which they are evaluated against pairs of execution states. This relational semantics directly expresses the comparison needed for noninterference. As a result, most of the complexities related to SecCSL's soundness are confined to the semantic level, whereas the calculus retains its similarity to standard Separation Logic and hence its simplicity.

Under this relational semantics (see Fig. 2 in Section 3), when a pair of states satisfies an assertion P, it implies that the two states agree on the values of all non-sensitive expressions as defined by P (Lemma 1). Noninterference is then stated as Theorem 2 on page 13: Program c with precondition P is secure against the ℓ_A-attacker if, whenever executed twice from two initial states jointly satisfying P and the lock invariants (and so agreeing on the values of all data assumed to be initially observable to the ℓ_A attacker), in all intermediate pairs of states arrived at after running each execution for the same number of steps, the resulting states again agree at that initially ℓ_A-visible memory. This definition is timing sensitive as it compares executions that have the same number of steps.

3 The Logic SecCSL

3.1 Assertions

Pure expressions e that do not depend on the heap are composed of variables x, function applications, equations, and conditional expressions. Pure relational formulas ρ comprise boolean expressions ϕ, value sensitivity $e :: e_l$, and relational implication $\Rightarrow$ (wlog. covering relational $\neg$, $\wedge$, $\vee$). We assume a standard first-order many sorted typing discipline (not elaborated).

$$e \ ::= \ x \mid f(e_1, \ldots, e_n) \mid e_1 = e_2 \mid \phi \ ? \ e_1 : e_2 \qquad \rho \ ::= \phi \mid e :: e_l \mid \rho_1 \Rightarrow \rho_2$$

We postulate that the logical signature contains a sort `Label`, corresponding to the security lattice, with constants $\texttt{low}, \texttt{high}\colon \texttt{Label}$ and a binary predicate symbol $\sqsubseteq\colon \texttt{Label} \times \texttt{Label} \to \texttt{Bool}$, whose interpretation satisfies the lattice axioms.

SecCSL's assertions P, Q may additionally refer to the heap and thus include the empty heap description, labelled points-to predicates (heap location sensitivity assertions), assertions guarded by (pure) conditionals, ordinary overlapping conjunction as well as separating conjunction, and existential quantification.

$$P \ ::= \ \rho \mid \texttt{emp} \mid e_p \overset{e_l}{\longmapsto} e_v \mid (\phi \ ? \ P : Q) \mid P \wedge Q \mid P \star Q \mid \exists \ x.\ P$$

Disjunction, negation, and implication are excluded because they cause issues for describing the set of ℓ-visible heap location to the ℓ-attacker, similarly to the problem of defining heap footprints for non-precise assertions [40,41,26]. These connectives can still occur between pure and relational expressions.

The standard expression semantics $[\![e]\!]_s$ evaluates e over a store s, which assigns values to variables x as $s(x)$. The interpretation $f^{\mathcal{A}}$ of a function symbol f is a function, given statically by a logical structure $\mathcal{A}$. Specifically, $\sqsubseteq^{\mathcal{A}}$ is the semantic ordering of the security lattice. We write $s \models \phi$ if $[\![\phi]\!]_s = \mathit{true}$.

The relational semantics of assertions, written $(s, h), (s', h') \models_\ell P$, is defined in Fig. 2 over two states (s, h) and (s', h') each consisting of a store and a heap. The semantics is defined against the attacker security level ℓ (called ℓ_A in Section 2.3). Stores s and s' are related via $e :: e_l$. We require the expression e_l denoting the sensitivity to coincide on s and s' and whenever $[\![e_l]\!]_s \sqsubseteq^{\mathcal{A}} \ell$ holds, e must evaluate to the same value both states, (7). Heaps are related by $(s, h), (s', h') \models_\ell e_p \overset{e_l}{\longmapsto} e_v$, which similarly ensures that the two heap fragments are identical $h = h'$ when e_l says so, (9). Conditional assertions $\phi \ ? \ P : Q$ evaluate to P when ϕ holds (relationally), and to Q otherwise. The separating conjunction splits both heaps independently, (12). Similarly, the existential quantifier picks two values v and v', (13). Whether parts of the split (resp. these two values) actually agree will depend on other assertions made.

To capture strong security properties, we require a declarative specification of which heap locations are considered visible to the ℓ-attacker, when assertion P holds in some (initial) state (see Section 2.3). We define this set in Fig. 3, denoted $\mathrm{lows}_\ell(P, s)$ for initial store s. Note that, by design, the definition does not give a useful result for existential like $\exists p\ v.\ p \overset{\texttt{low}}{\longmapsto} v$. This mirrors the usual difficulty

Using the abbreviation $s, h \models e_p \mapsto e_v \iff h = \{[\![e_p]\!]_s \mapsto [\![e_v]\!]_s\}$

$$(s,h),(s',h') \models_\ell \texttt{emp} \iff h = h' = \varnothing \tag{5}$$

$$(s,h),(s',h') \models_\ell \phi \iff s \models \phi \text{ and } s' \models \phi \tag{6}$$

$$\begin{aligned}&(s,h),(s',h') \models_\ell e :: e_l \\ &\quad\iff [\![e_l]\!]_s = [\![e_l]\!]_{s'} \text{ and } \left([\![e_l]\!]_s \sqsubseteq^{\mathcal{A}} \ell \implies [\![e]\!]_s = [\![e]\!]_{s'}\right)\end{aligned} \tag{7}$$

$$\begin{aligned}&(s,h),(s',h') \models_\ell \rho_1 \Rightarrow \rho_2 \\ &\quad\iff (s,h),(s',h') \models_\ell \rho_1 \text{ implies } (s,h),(s',h') \models_\ell \rho_2\end{aligned} \tag{8}$$

$$\begin{aligned}&(s,h),(s',h') \models_\ell e_p \overset{e_l}{\longmapsto} e_v \\ &\quad\iff s,h \models e_p \mapsto e_v \text{ and } s',h' \models e_p \mapsto e_v \text{ and } (s,h),(s',h') \models e_p :: e_l \wedge e_v :: e_l\end{aligned} \tag{9}$$

$$\begin{aligned}&(s,h),(s',h') \models_\ell (\phi \ ?\ P : Q) \\ &\quad\iff \begin{cases}(s,h),(s',h') \models_\ell P, & \text{if } s \models \phi \text{ and } s' \models \phi \\ (s,h),(s',h') \models_\ell Q, & \text{otherwise}\end{cases}\end{aligned} \tag{10}$$

$$\begin{aligned}&(s,h),(s',h') \models_\ell P \wedge Q \\ &\quad\iff (s,h),(s',h') \models_\ell P \text{ and } (s,h),(s',h') \models_\ell Q\end{aligned} \tag{11}$$

$$\begin{aligned}&(s,h),(s',h') \models_\ell P \star Q \\ &\quad\iff \text{there are disjoint sub-heaps } h_1, h_2 \text{ and } h_1', h_2' \\ &\qquad\quad \text{with } h = h_1 \uplus h_2 \text{ and } h' = h_1' \uplus h_2' \\ &\qquad\quad \text{such that } (s,h_1),(s',h_1') \models_\ell P_1 \text{ and } (s,h_2),(s',h_2') \models_\ell P_2\end{aligned} \tag{12}$$

$$\begin{aligned}&(s,h),(s',h') \models_\ell \exists\ x.\ P \\ &\quad\iff \text{there are values } v, v' \text{such that } (s(x \mapsto v),h),(s'(x \mapsto v'),h') \models P\end{aligned} \tag{13}$$

Fig. 2. Relational semantics of assertions.

$$\begin{aligned}\text{lows}_\ell(\rho, s) &= \emptyset, \qquad \text{notably } \text{lows}_\ell(e :: e_l, s) = \emptyset \\ \text{lows}_\ell(P \star Q, s) &= \text{lows}_\ell(P \wedge Q, s) = \text{lows}_\ell(P, s) \cup \text{lows}_\ell(Q, s) \\ \text{lows}_\ell(e_p \overset{e_l}{\longmapsto} e_v, s) &= \begin{cases}\{[\![e_p]\!]_s\}, & [\![e_l]\!]_s \sqsubseteq^{\mathcal{A}} \ell \\ \emptyset, & \text{otherwise}\end{cases} \\ \text{lows}_\ell(\phi \ ?\ P : Q, s) &= \begin{cases}\text{lows}_\ell(P, s), & s \models \phi \\ \text{lows}_\ell(Q, s), & \text{otherwise}\end{cases} \\ \text{lows}_\ell(\exists\ x.\ P, s) &= \begin{cases}\text{lows}_\ell(P, s), & \forall\ v.\ \text{lows}_\ell(P, s) = \text{lows}_\ell(P, s(x \mapsto v)) \\ \emptyset, & \text{otherwise}\end{cases}\end{aligned}$$

Fig. 3. Low locations of an assertion.

of defining footprints for non-precise separation logic assertions [40,41,26]. This restriction is not an issue in practice, as location sensitivity assertions $e_p \overset{e_l}{\longmapsto} e_v$ are intended to describe the static regions of memory (data sinks) visible to the attacker, for which existential quantification over variables free in e_p or e_l is neither useful nor necessary.

3.2 Entailments

Although implications between spatial formulas is not part of the assertion language, entailments $P \overset{\ell}{\Longrightarrow} Q$ between assertions still play a role in SecCSL's Hoare style consequence rule (Conseq in Fig. 4). We discuss entailment now as it sheds useful light on some consequences of SecCSL's relational semantics.

Definition 1 (Secure Entailment). $P \overset{\ell}{\Longrightarrow} Q$ *holds iff*

- $(s,h),(s',h') \models_\ell P$ *implies* $(s,h),(s',h') \models_\ell Q$ *for all* s,h *and* s',h', and
- $\mathrm{lows}_\ell(P,s) \subseteq \mathrm{lows}_\ell(Q,s)$ *for all* s

The security level ℓ is used not just in the evaluation of the assertions but also to preserve the ℓ-attacker visible locations of P in Q. This reflects the intuition that P is stronger than Q, and so Q should make fewer assumptions than P on the limitations of an attacker's observational powers.

Proposition 1.

$$e = e' \wedge e_l = {e_l}' \wedge e :: e_l \overset{\ell}{\Longrightarrow} e' :: {e_l}' \tag{14}$$

$$e :: e_l \wedge e_l \sqsubseteq {e_l}' \wedge {e_l}' :: \ell \overset{\ell}{\Longrightarrow} e :: {e_l}' \tag{15}$$

$$e_l :: \ell \overset{\ell}{\Longrightarrow} c :: e_l \qquad \textit{for a constant } c \tag{16}$$

$$e_1 :: e_l \wedge \cdots \wedge e_n :: e_l \overset{\ell}{\Longrightarrow} f(e_1,\ldots,e_n) :: e_l \qquad \textit{for } n > 0 \tag{17}$$

$$e_p \overset{e_l}{\longmapsto} e_v \wedge e_l \sqsubseteq \ell \overset{\ell}{\Longrightarrow} e_p \overset{e_l}{\longmapsto} e_v \wedge e_p :: e_l \wedge e_v :: e_l \tag{18}$$

$$\big(\forall\ s.\ \mathrm{lows}_\ell(P,s) = \mathrm{lows}_\ell(Q,s)\big) \ \textit{ implies } \ \phi \wedge (\phi\ ?\ P : Q) \overset{\ell}{\Longrightarrow} P \tag{19}$$

$$P \overset{\ell}{\Longrightarrow} P' \textit{ and } Q \overset{\ell}{\Longrightarrow} Q' \ \textit{ implies } \ P \star Q \overset{\ell}{\Longrightarrow} P' \star Q' \tag{20}$$

Entailment (14) in Proposition 1 shows that sensitivity of values is compatible with equality. This property fails in the security separation logic of [14], where labels are part of the semantics of expressions but are not compared by equality. The second property (15) captures the intuition that less-sensitive data can always be used in contexts where more-sensitive data might be expected (but not vice-versa). Recall that ${e_l}'$ here is an expression. The additional condition ${e_l}' :: \ell$ guarantees that this expression denotes a meaningful security level, i.e. evaluates identically in both states (cf. (7)). (abusing notation to let the semantic ℓ stand for some expression that denotes it). Property (16) encodes that constants do not depend on any state; again the security level expression e_l must be meaningful,

but trivially $c :: \ell$ when ℓ is constant, too. Value sensitivity is congruent with function application (17). This is not surprising, as functions map arguments equal in both states to equal results. Yet, as with (14) above, this property fails in [14] where security labels are attached to values. Note that the reverse entailment is false (e.g. for the constant function $\lambda x.c$).

Via (18), when $e_p \overset{e_l}{\longmapsto} e_v$ it follows that both the location e_p and the value e_v adhere to the level e_l, cf. (9). Note that the antecedent $e_p \overset{e_l}{\longmapsto} e_v$ is repeated in the consequent to ensure that the set of ℓ-attacker visible locations is preserved. Conditional assertions can be resolved when the test is definite, provided that P and Q describe the same set of public locations, (19) and symmetrically for $\neg\phi$.

3.3 Proof System

We consider a canonical concurrent programming language with shared heap locations protected by locks but without shared variables. Commands c comprise assignments to local variables, heap access (load and store),[3] standard sequential programming constructs, as well as parallel composition and locking. We assume a static collection of valid lock identifiers l, each of which has an assertion as its associated invariant $\mathrm{inv}(l)$, describing the protected portion of the heap. We postpone the (fairly standard) semantics to Section 4 where we present the soundness proof.

$$\begin{aligned} c \;::=\;& x := e \mid x := [e_v] \mid [e_p] := e_v \mid \texttt{lock}\ l \mid \texttt{unlock}\ l \\ & c_1; c_2 \mid c_1 \parallel c_2 \mid \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ b\ \texttt{do}\ c \end{aligned}$$

The SecCSL proof rules are shown in Fig. 4. They extend the standard rules of concurrent separation logic [38] (CSL) by additional side-conditions that amount to information flow checks $e :: \_$ as part of the respective preconditions.

Similarly to [46], without loss of generality we require that assignments (rules Asg, Load) are always to distinct variables, to avoid renaming in the assertions. Storing to a heap location through an e_l-sensitive location $e_p \overset{e_l}{\longmapsto} e_v$ (rule Store) requires that the value e_v written to that location admits the corresponding security level e_l of the location e_p. Note that due to monotonicity (15) the security level does not have to match exactly. The rules for locking are standard [12]. To preclude information leaks through timing channels, the execution can branch on non-secret values only; similarly for logical case distinctions. This manifests in side conditions $b :: \ell$ for the respective branching condition b where, recall, ℓ is the attacker security level (rules If, Split, While). The consequence rule (Conseq) uses entailment relative to ℓ (Definition 1). Rule Par has the usual proviso that the variables modified in one thread cannot interfere with those relied on by the other and its pre-/postcondition.

[3] Volatile memory locations can be treated analogously to locks by introducing an additional assertion characterizing that part of the heap, that is implicitly available to atomic commands. This feature is realized in the Isabelle theories [18] but omitted here in the interests of brevity.

$$\frac{x \notin \text{free}(e)}{\ell \vdash \{\texttt{emp}\}\ x := e\ \{x = e\}}\ \textsc{Asg} \qquad \frac{x \notin \text{free}(e_p, e_v, e_l)}{\ell \vdash \{e_p \overset{e_l}{\mapsto} e_v\}\ x := [e_p]\ \{x = e_v \wedge e_p \overset{e_l}{\mapsto} e_v\}}\ \textsc{Load}$$

$$\frac{}{\ell \vdash \{e_v :: e_l \wedge e_p \overset{e_l}{\mapsto} \_\}\ [e_p] := e_v\ \{e_p \overset{e_l}{\mapsto} e_v\}}\ \textsc{Store}$$

$$\frac{}{\ell \vdash \{\texttt{emp}\}\ \texttt{lock}\ l\ \{\text{inv}(l)\}}\ \textsc{Lock} \qquad \frac{}{\ell \vdash \{\text{inv}(l)\}\ \texttt{unlock}\ l\ \{\texttt{emp}\}}\ \textsc{Unlock}$$

$$\frac{\ell \vdash \{b \wedge P\}\ c\ \{Q\} \quad \ell \vdash \{\neg b \wedge P\}\ c\ \{Q\}}{\ell \vdash \{b :: \ell \wedge P\}\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{Q\}}\ \textsc{If} \qquad \frac{\ell \vdash \{b \wedge P\}\ c\ \{Q\} \quad \ell \vdash \{\neg b \wedge P\}\ c\ \{Q\}}{\ell \vdash \{b :: \ell \wedge P\}\ c\ \{Q\}}\ \textsc{Split}$$

$$\frac{\ell \vdash \{b \wedge (b :: \ell) \wedge P\}\ c\ \{b :: \ell \wedge P\}}{\ell \vdash \{b :: \ell \wedge P\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{\neg b \wedge P\}}\ \textsc{While}$$

$$\frac{\ell \vdash \{P\}\ c_1\ \{R\} \quad \ell \vdash \{R\}\ c_2\ \{Q\}}{\ell \vdash \{P\}\ c_1; c_2\ \{Q\}}\ \textsc{Seq} \qquad \frac{\text{modified}(c) \cap \text{free}(F) = \varnothing \quad \ell \vdash \{P\}\ c\ \{Q\}}{\ell \vdash \{P \star F\}\ c\ \{Q \star F\}}\ \textsc{Frame}$$

$$\frac{P \overset{\ell}{\Longrightarrow} P' \quad Q' \overset{\ell}{\Longrightarrow} Q \quad \ell \vdash \{P'\}\ c\ \{Q'\}}{\ell \vdash \{P\}\ c\ \{Q\}}\ \textsc{Conseq} \qquad \frac{\text{modified}(c_i) \cap \text{free}(c_j, P_j, Q_j) = \varnothing \text{ for } i \neq j \quad \ell \vdash \{P_1\}\ c_1\ \{P_1\} \quad \ell \vdash \{P_2\}\ c_2\ \{P_2\}}{\ell \vdash \{P_1 \star P_2\}\ c_1 \parallel c_2\ \{Q_1 \star Q_2\}}\ \textsc{Par}$$

Fig. 4. Proof Rules of SecCSL.

4 Security Definition & Soundness

The soundness theorem for SecCSL guarantees that if some triple $\ell \vdash \{P\}\ c\ \{Q\}$ is derived using the rules of Fig. 4, then: all executions of c started in a state satisfying precondition P are memory *safe*, partially *correct* with respect to postcondition Q, and moreover *secure* with respect to the sensitivity of values as denoted by P and Q and at all times respect the sensitivity of locations as denoted by P (see Section 2.3). Proof outlines are relegated to Appendix B. All results have been mechanised in Isabelle/HOL [37] and are available at [18].

The top-level security property of SecCSL is a noninterference condition [19]. Noninterference as a security property specifies, roughly, that for any pair of executions that start in states that agree on the values of all attacker-observable inputs, then, from the attacker's point of view the resulting executions will be indistinguishable, i.e. all of the attacker visible observations will agree. In SecCSL, what is "attacker-observable" depends on the attacker level ℓ. The "inputs" are the expressions e, and the attacker-visible inputs are those expressions e whose sensitivity is given by $e :: \ell'$ judgements in the precondition P for which $\ell' \sqsubseteq \ell$. The attacker-visible observations are the contents of all memory locations in

$\text{lows}_\ell(P, s)$, for initial store s and precondition P. Thus we define when two heaps are indistinguishable to the ℓ-attacker.

Definition 2 (ℓ Equivalence). *Two heaps coincide on a set of locations A, written $h \equiv_A h'$, iff for all $a \in A$. $a \in \text{dom}\ (h) \cap \text{dom}\ (h')$ and $h(a) = h'(a)$. Two heaps h and h' are ℓ-equivalent wrt. store s and assertion P, if $h \equiv_A h'$ for $A = \text{lows}_\ell(P, s)$.*

Then, the ℓ-validity of an assertion P in the relational semantics witnesses ℓ-equivalence between the corresponding heaps.

Lemma 1. *If $(s, h), (s', h') \models_\ell P$, then $h \equiv_A h'$ for $A = \text{lows}_\ell(P, s)$.*

Furthermore, if $(s, h), (s', h') \models_\ell P$, then $\text{lows}_\ell(P, s) = \text{lows}_\ell(P, s')$ since the security levels in labeled points-to predicates must coincide on s and s', cf. (9).

Semantics. Semantic configurations, denoted by k in the following, are one of three kinds: $(\textbf{run}\ c, L, s, h)$ denotes a command c in a state s, h where L is a set of locks that are currently not held by any thread and can be acquired by c; $(\textbf{stop}\ L, s, h)$ similarly denotes a final state s, h with residual locks L, and **abort** results from invalid heap access.

The single-step relation $(\textbf{run}\ c, L, s, h) \xrightarrow{\sigma} k$ takes running configurations to successors k with respect to a schedule σ that resolves the non-determinism of parallel composition. The schedule σ is a list of *actions*: the action $\langle\tau\rangle$ represents the execution of atomic commands and the evaluation of conditionals; the actions $\langle 1\rangle$ and $\langle 2\rangle$ respectively denote the execution of the left- and right-hand sides of a parallel composition for a single step, and so define a deterministic scheduling discipline reminiscent of separation kernels [32]. For example, $(\textbf{run}\ c_1 \parallel c_2, L, s, h) \xrightarrow{\langle 1\rangle\cdot\sigma} (\textbf{run}\ c_1' \parallel c_2, L', s', h')$ if $(\textbf{run}\ c_1, L, s, h) \xrightarrow{\sigma} (\textbf{run}\ c_1', L', s', h')$. Configurations $(\textbf{run}\ \texttt{lock}\ l, L, s, h)$ can only be scheduled if $l \in L$ (symmetrically for `unlock`)) and otherwise block without a possible step.

Executions $k_1 \xrightarrow{\sigma_1 \cdots \sigma_n}{}^* k_{n+1}$ chain several steps $k_i \xrightarrow{\sigma_i} k_{i+1}$ by accumulating the schedule. We are considering partial correctness only, thus the schedule is always finite and so are all executions. The rules for program steps are otherwise standard and can be found in Appendix A.

Compositional Security. To prove that SecCSL establishes its top-level non-interference condition, we first define a compositional security condition that provides the central characterization of security for a command c with respect to precondition P and postcondition Q. That central, compositional property we denote $\text{secure}_\ell^n(P, c, Q)$ and formalize below in Definition 3. It ensures that the first n steps (or fewer if the program terminates before that) are safe and preserve ℓ-equivalence of the heap locations specified initially in P, but in a way that is compositional across multiple execution steps, across multiple threads of execution and across different parts of the heap. It is somewhat akin, although more precise than, prior characterizations based on *strong low bisimulation* [45,16].

Disregarding the case when c terminates before the n-th step for a moment, for a pair of initial states (s_1, h_1) and (s_1', h_1') and initial set of locks L_1, and

a fixed schedule $\sigma = \sigma_1 \cdots \sigma_n$, $\mathrm{secure}_\ell^{n+1}(P_1, c_1, Q)$ requires that c performs a sequence of lockstep execution steps from each initial state

$$(\mathbf{run}\ c_i, L_i, s_i, h_i) \xrightarrow{\sigma_i} (\mathbf{run}\ c_{i+1}, L_{i+1}, s_{i+1}, h_{i+1}) \qquad \text{for } 1 \le i \le n \qquad (21)$$
$$(\mathbf{run}\ c_i, L_i, s'_i, h'_i) \xrightarrow{\sigma_i} (\mathbf{run}\ c_{i+1}, L_{i+1}, s'_{i+1}, h'_{i+1})$$

These executions must agree on the intermediate commands c_i and locks L_i and the ith pair of states must satisfy an intermediate assertion of the following form:

$$(s_i, h_i), (s'_i, h'_i) \models_\ell P_i \star F \star \mathrm{invs}(L_i) \quad \text{where } \mathrm{invs}(L_i) = \bigstar_{l_i \in L_i} \mathrm{inv}(l_i) \quad (22)$$

Here P_i describes the part of the heap that command c_i is currently accessing. $\mathrm{invs}(L_i)$ is the set of lock invariants for the locks $l_i \in L_i$ not currently acquired. Its presence ensures that whenever a lock is acquired that the associated invariant can be assumed to hold. Finally F is an arbitrary *frame*, an assertion that does not mention variables updated by c_i. Its inclusion allows the security property to compose with respect to different parts of the heap.

Moreover, each $P_{i+1} \star \mathrm{invs}(L_{i+1})$ is required to preserve the sensitivity of all ℓ-visible heap locations of $P_i \star \mathrm{invs}(L_i)$, i.e. so that $\mathrm{lows}_\ell(P_i \star \mathrm{invs}(L_i), s_i) \subseteq \mathrm{lows}_\ell(P_{i+1} \star \mathrm{invs}(L_{i+1}), s_{i+1})$. If some intermediate step $m \le n$ terminates, then $P_{m+1} = Q$, ensuring the postcondition holds when the executions terminate. Lastly, neither execution is allowed to reach an **abort** configuration.

If the initial state satisfies $P_1 \star F \star \mathrm{invs}(L_1)$ then (22) holds throughout the entire execution, and establishes the end-to-end property that any final state indeed satisfies the postcondition and that $\mathrm{lows}_\ell(P_1 \star \mathrm{invs}(L_1), s_1) \subseteq \mathrm{lows}_\ell(P_i \star \mathrm{invs}(L_i), s_i)$ with respect to the initially specified low locations.

The property $\mathrm{secure}_\ell^n(P, c, Q)$ is defined recursively to match the steps of the lockstep execution of the program.

Definition 3 (Security).

- $\mathrm{secure}_\ell^0(P_1, c_1, Q)$ *holds always.*
- $\mathrm{secure}_\ell^{n+1}(P_1, c_1, Q)$ *holds, iff for all pairs of states* (s_1, h_1), (s'_1, h'_1), *frames* F, *and sets of locks* L_1, *such that* $(s_1, h_1), (s'_1, h'_1) \models_\ell P_1 \star F \star \mathrm{invs}(L_1)$, *and given two steps* $(\mathbf{run}\ c_1, L_1, s_1, h_1) \xrightarrow{\sigma} k$ *and* $(\mathbf{run}\ c_1, L_1, s'_1, h'_1) \xrightarrow{\sigma} k'$ *there exists an assertion* P_2 *and a pair of successor states with either of*
  - $k = (\mathbf{stop}\ L_2, s_2, h_2)$ *and* $k' = (\mathbf{stop}\ L_2, s'_2, h'_2)$ *and* $P_2 = Q$
  - $k = (\mathbf{run}\ c_2, L_2, s_2, h_2)$ *and* $k' = (\mathbf{run}\ c_2, L_2, s'_2, h'_2)$ *with* $\mathrm{secure}_\ell^n(P_2, c_2, Q)$

  such that $(s_2, h_2), (s'_2, h'_2) \models_\ell P_2 \star F \star \mathrm{invs}(L_2)$ *and* $\mathrm{lows}_\ell(P_1 \star \mathrm{invs}(L_1), s_1) \subseteq \mathrm{lows}_\ell(P_2 \star \mathrm{invs}(L_2), s_2)$ *in both cases.*

Two further side condition are imposed, ensuring all mutable shared state lies in the heap (cf. Section 3): c_1 doesn't modify variables occurring in $\mathrm{invs}(L_1)$ and F (which guarantees that both remain intact), and the free variables in P_2 can only mention those already present in P_1, c_1, or in any lock invariant (which guarantees that P_2 remains stable against concurrent assignments). Note that each step can pick a different frame F, required for the soundness of rule Par.

Lemma 2. $\ell \vdash \{P\}\ c\ \{Q\}$ *implies* $\mathrm{secure}_\ell^n(P, c, Q)$ *for every* $n \ge 0$.

Safety, Correctness and Noninterference. Execution safety and correctness with respect to pre- and postcondition follow straightforwardly from Lemma 2.

Corollary 1 (Safety). *Given initial states* $(s_1, h_1), (s_1', h_1') \models_\ell P \star \text{invs}(L_1)$ *and two executions of a command* c *under the same schedule to resulting configurations* k *and* k' *respectively, then* $\ell \vdash \{P\}\ c\ \{Q\}$ *implies* $k \neq \textbf{abort} \wedge k' \neq \textbf{abort}$.

Theorem 1 (Correctness). *For initial states* $(s_1, h_1), (s_1', h_1') \models_\ell P \star \text{invs}(L_1)$, *given two complete executions of a command* c *under the same schedule* σ

$$(\textbf{run}\ c, L_1, s_1, h_1) \xrightarrow{\sigma}{}^* (\textbf{stop}\ L_2, s_2, h_2)$$
$$(\textbf{run}\ c_i, L_i, s_i', h_i') \xrightarrow{\sigma}{}^* (\textbf{stop}\ L_2, s_2', h_2')$$

then $\ell \vdash \{P\}\ c\ \{Q\}$ *implies* $(s_2, h_2), (s_2', h_2') \models_\ell Q \star \text{invs}(L_2)$.

The top-level noninterference property also follows from Lemma 2 via Lemma 1. For brevity, we state the noninterference property directly in the theorem:

Theorem 2 (Noninterference). *Given a command* c, *and initial states* $(s_1, h_1), (s_1', h_1') \models_\ell P \star \text{invs}(L_1)$ *then* $\ell \vdash \{P\}\ c\ \{Q\}$ *implies* $h_i \equiv_A h_i'$, *where* $A = \text{lows}_\ell(P \star \text{invs}(L_1), s_1)$, *for all pairs of heaps* h_i *and* h_i' *arising from executing the same schedule from each initial state.*

5 SecC: Automating SecCSL

To demonstrate the ease by which SecCSL can be automated, we implemented SecC, available at [18]. SecC implements the logic from Section 3 for a subset of C. SecC is currently used to explore reasoning about example programs with interesting security policies. Thus its engineering has focused on features related to security reasoning (e.g. deciding when conditions $e :: e_l$ are entailed) rather than reasoning about complex data structures.

Symbolic Execution. SecC automates SecCSL through symbolic execution, as pioneered for SL in [7]. Similarly to VeriFast's algorithm in [22], the verifier computes the strongest postcondition of a command c when executed in a symbolic state, yielding a set of possible final symbolic states. Each such state $\sigma = (\rho, s, \underline{P})$ maintains a path condition ρ of relational formulas (from procedure contracts, invariants, and the evaluation of conditionals) and a symbolic heap described by a list $\underline{P} = (P_1 \star \cdots \star P_n)$ of atomic spatial assertions (points-to and instances of defined predicates). The symbolic store s maps program variables to pure expressions, where $s(e)$ denotes substituting s into e. As an example, when $P_i = s(e_p) \mapsto v$ is part of the symbolic heap, a load $x := e_p$ in σ can be executed to yield the updated state $(\rho, s(x := v), \underline{P})$ where x is mapped to v.

To find the P_i we match the left-hand sides of points-to predicates. Similarly, matching is used during checking of entailments $\rho_1 \wedge \underline{P} \overset{\ell}{\Longrightarrow} \exists\ \underline{x}.\ \rho_2 \wedge \underline{Q}$, where the conclusion is normalized to prenex form. The entailment is reduced

to a non-spatial problem by incrementally computing a substitution τ for the existentials $\underline{x}$, removing pairs $P_i = \tau(Q_j)$ in the process, as justified by (20) (see also "subtraction rules" in [7, Sec. 4]).

Finally, the remaining relational problem $\rho_1 \Rightarrow \rho_2$ without spatial connectives can be encoded into first-order [17], by duplicating the pure formulas in terms of fresh variables to represent the second state, and by the syntactic equivalent of (7). The resulting verification condition is discharged with Z3 [15]. This translation is semantically complete. For example, consider Fig. 4 from Prabawa et al. [43]. It has a conditional `if(b == b) ...`, whose check $(b = b) :: \texttt{low}$, translated to $(b = b) = (b' = b')$ by SecC, holds independently of b's sensitivity.

Features. In addition to the logic from Section 3, SecC supports procedure modular verification with pre-/postconditions as usual; and it supports user-defined spatial predicates. While some issues of the C source language are not addressed (yet), such as integer overflow, those that impact directly on information flow security are taken into account. Specifically, the shortcut semantics of boolean operators `&&`, `||`, and ternary `_ ? _ : _` count as branching points and as such the left hand side resp. the test must not depend on sensitive data, similarly to the conditions of `if` statements and `while` loops.

A direct benefit of the integration of security levels into the assertion language is that it becomes possible to specify the sensitivity of data passed to library and operating system functions. For example, the execution time of `malloc(len)` would depend on the value of `len`, which can thus be required to satisfy `len` $::$ `low` by annotating its function header with an appropriate precondition, using SecC's `requires` annotation. Likewise, SecC can reason about limited forms of declassification, in which external functions are trusted to safely release otherwise sensitive data, by giving them appropriate pre-/postconditions. For example, a password hashing library function prototype might be annotated with a postcondition asserting its result is `low`, via SecC's `ensures` annotation.

Examples and Case Study. SecC proves Fig. 1 secure, and correctly flags buggy variants as insecure, e.g., where the test in thread 1 is reversed, or when thread 2 does not clear the `data` field upon setting the `is_classified` to `FALSE`. SecC also correctly analyzes those 7 examples from [17] that are supported by the logic and tool (each in ∼10 milliseconds). All examples are available at [18].

To compare SecC and SecCSL against the recent Covern logic [34], we took a non-trivial example program that Murray et al. verified in Covern, manually translated it to C, and verified it automatically using SecC. The original program[4], written in Covern's tiny While language embedded in Isabelle/HOL, models the software functionality of a simplified implementation of the Cross Domain Desktop Compositor (CDDC) [5]. The CDDC is a device that facilitates interactions with multiple PCs, each of which runs applications at differing sensitivity, from a single keyboard, mouse and display. Its multi-threaded software

[4] `https://bitbucket.org/covern/covern/src/master/examples/cddc/Example_CDDC_WhileLockLanguage.thy`

handles routing of keyboard input to the appropriate PC and switching between the PCs via mouse gestures. Verifying the C translation required adding SecCSL annotations for procedure pre-/postconditions and loop invariants. The C translation including those annotations is ∼250 lines in length. The present, unoptimised, implementation of SecC verifies the resulting artifact in ∼5 seconds. In contrast, the Covern proof of this example requires ∼600 lines of Isabelle/HOL definitions/specification, plus ∼550 lines of Isabelle proof script.

6 Related Work

There has been much work targeting type systems and program logics for concurrent information flow. Karbyshev et al. [23] provide an excellent overview. Here we concentrate on work whose ideas are most closely related to SecCSL.

Costanzo and Shao [14] propose a sequential separation logic for reasoning about information flow. Unlike SecCSL, theirs does not distinguish value and location sensitivity. Their separation logic assertions have a fairly standard (non-relational) semantics, at the price of having a *security-aware* language semantics that propagates security labels attached to values in the store and heap. As mentioned in Section 3.2, this has the unfortunate side-effect of breaking intuitive properties about sensitivity assertions. We conjecture that the absence of such properties would make their logic harder to automate than SecCSL, which SecC demonstrates is feasible. SecCSL avoids the aforementioned drawbacks by adopting a relational assertion semantics.

Gruetter and Murray [20] propose a security separation logic in Coq [8] for Verifiable C, the C subset of the Verified Software Toolchain [3,2]. However they provide no soundness proof for its rules and its feasibility to automate is unclear.

Two recent compositional logics for concurrent information flow are the Covern logic [34] and the type and effect system of Karbyshev et al. [23]. Both borrow ideas from separation logic. However, unlike SecCSL, neither is defined for languages with pointers, arrays etc.

Like SecCSL, Covern proves a timing-sensitive security property. Location sensitivity is defined statically by value-dependent predicates, and value sensitivity is tracked by a dependent security typing context Γ [35], relative to a Hoare logic predicate P over the entire shared memory. In Covern locks carry non-relational invariants. In contrast, SecCSL unifies these elements together into separation logic assertions with a relational semantics. Doing so leads to a much simpler logic, amenable to automation, while supporting pointers, etc.

On the other hand, Karbyshev et al. [23] prove a timing-*insensitive* security property, but rely on primitives to interact with the scheduler to prevent leaks via scheduling decisions. Unlike SecCSL, which assumes a deterministic scheduling discipline, Karbyshev et al. support a wider class of scheduling policies. Their system tracks resource ownership and transfer between threads at synchronisation points, similar to CSLs. Their resources include *labelled scheduler resources* that account for scheduler interaction, including when scheduling decisions become

tainted by secret data—something that cannot occur in SecCSL's deterministic scheduling model.

Prior logics for sequential languages, e.g. [1,4], have also adopted separation logic ideas to reason locally about memory, combining them with relational assertions similar to SecCSL's $e :: e_l$ assertions. For instance, the agreement assertions $\mathsf{A}(e)$ of [4] coincide with SecCSL's $e :: \texttt{low}$. Unlike SecCSL, some of these logics support languages with explicit declassification actions [4].

Self-composition is another technique to exploit existing verification infrastructure for proofs of general hyperproperties [13], including but not limited to non-interference. Eilers et al. [17] present such an approach for Viper, which supports an assertion language similar to that of Separation Logic. It does not support public heap locations (which are information sources and sinks at the same time) albeit sinks can be modeled via preconditions of procedures. A similar approach is implemented in Frama-C [9]. Both of [17,9] do not support concurrency, and it remains unclear how self-composition could avoid an exponential blow-up from concurrent interleaving, which SecCSL avoids.

The soundness proof for SecCSL follows the general structure of Vafeiadis' [46] for CSL, which is also mechanised in Isabelle/HOL. There is, however, a technical difference: His analog of Definition 3, a recursive predicate called $\text{safe}_n(c, s, h, Q)$, refers to a semantic initial state s, h whereas we propagate a syntactic assertion (22) only. Our formulation has the benefit that some of the technical reasoning in the soundness proof is easier to automate. Its drawback is the need to impose technical side-conditions on the free variables of the frame F and the intermediate assertions P_i.

7 Conclusion

We presented SecCSL, a concurrent separation logic for proving expressive data-dependent information flow properties of programs. SecCSL is considerably simpler, yet handles features like pointers, arrays etc., which are out of scope for contemporary logics. It inherits the structure of traditional concurrent separation logics, and so like those logics can be automated via symbolic execution [22,30,10]. To demonstrate this, we implemented SecC, an automatic verifier for expressive information flow security for a subset of the C language.

Separation logic has proved to be a remarkably powerful vehicle for reasoning about programs, weak memory concurrency [47], program synthesis [42], and many other domains. With SecCSL, we hope that in future the same possibilities might be opened to verified information flow security.

Acknowledgement. We thank the anonymous reviewers for their careful and detailed comments that helped significantly to clarify the discussion of finer points.

This research was sponsored by the Department of the Navy, Office of Naval Research, under award #N62909-18-1-2049. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Office of Naval Research.

References

1. Amtoft, T., Bandhakavi, S., Banerjee, A.: A logic for information flow in object-oriented programs. In: Proc. of Principles of Programming Languages (POPL). pp. 91–102. ACM (2006)
2. Appel, A.W., Dockins, R., Hobor, A., Beringer, L., Dodds, J., Stewart, G., Blazy, S., Leroy, X.: Program Logics for Certified Compilers. Cambridge University Press, New York, NY, USA (2014)
3. Appel, A.W., others: The Verified Software Toolchain. `https://github.com/PrincetonUniversity/VST` (2017)
4. Banerjee, A., Naumann, D.A., Rosenberg, S.: Expressive declassification policies and modular static enforcement. In: Proc. of Symposium on Security and Privacy (S&P). pp. 339–353. IEEE (2008)
5. Beaumont, M., McCarthy, J., Murray, T.: The Cross Domain Desktop Compositor: using hardware-based video compositing for a multi-level secure user interface. In: Annual Computer Security Applications Conference (ACSAC). pp. 533–545. ACM (2016)
6. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Proc. of Principles of Programming Languages (POPL). pp. 14–25. ACM (2004)
7. Berdine, J., Calcagno, C., O'hearn, P.W.: Symbolic execution with Separation Logic. In: Asian Symposium on Programming Languages and Systems (APLAS). pp. 52–68. Springer (2005)
8. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004)
9. Blatter, L., Kosmatov, N., Le Gall, P., Prevosto, V., Petiot, G.: Static and dynamic verification of relational properties on self-composed C code. In: International Conference on Tests and Proofs. pp. 44–62. Springer (2018)
10. Calcagno, C., Distefano, D.: Infer: An automatic program verifier for memory safety of C programs. In: NASA Formal Methods Symposium. pp. 459–465. Springer (2011)
11. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: NASA Formal Methods Symposium. pp. 3–11. Springer (2015)
12. Chlipala, A.: Formal Reasoning About Programs (2016)
13. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: Proc. of Computer Security Foundations Symposium (CSF). pp. 51–65 (2008)
14. Costanzo, D., Shao, Z.: A separation logic for enforcing declarative information flow control policies. In: Proc. of Principles of Security and Trust (POST). LNCS, vol. 8414, pp. 179–198. Springer (2014)
15. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 337–340. Springer (2008)
16. Del Tedesco, F., Sands, D., Russo, A.: Fault-resilient non-interference. In: Proc. of Computer Security Foundations Symposium (CSF). pp. 401–416. IEEE (2016)
17. Eilers, M., Müller, P., Hitz, S.: Modular product programs. In: Proc. of European Symposium on Programming (ESOP). pp. 502–529. Springer (2018)

18. Ernst, G., Murray, T.: SecC tool description and Isabelle theories for SecCSL. https://covern.org/secc (2019)
19. Goguen, J., Meseguer, J.: Security policies and security models. In: Proc. of Symposium on Security and Privacy (S&P). pp. 11–20. Oakland, California, USA (Apr 1982)
20. Gruetter, S., Murray, T.: Short paper: Towards information flow reasoning about real-world C code. In: Proc. of Workshop on Programming Languages and Analysis for Security (PLAS). pp. 43–48. ACM (2017)
21. Gu, R., Shao, Z., Chen, H., Wu, X.N., Kim, J., Sjöberg, V., Costanzo, D.: CertiKOS: An extensible architecture for building certified concurrent OS kernels. In: Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Nov 2016)
22. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: NASA Formal Methods Symposium. pp. 41–55. Springer (2011)
23. Karbyshev, A., Svendsen, K., Askarov, A., Birkedal, L.: Compositional noninterference for concurrent programs via separation and framing. In: Proc. of Principles of Security and Trust (POST) (2018)
24. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Transactions on Computer Systems **32**(1), 2:1–2:70 (Feb 2014)
25. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7), 107–115 (2009)
26. Löding, C., Madhusudan, P., Murali, A., na, L.P.: A first order logic with frames, under submission. Available at: http://madhu.cs.illinois.edu/FOFrameLogic.pdf
27. Lourenço, L., Caires, L.: Dependent information flow types. In: Proc. of Principles of Programming Languages (POPL). pp. 317–328. Mumbai, India (Jan 2015)
28. Mantel, H., Sands, D.: Controlled declassification based on intransitive noninterference. In: Asian Symposium on Programming Languages and Systems (APLAS). pp. 129–145. Springer (2004)
29. Mantel, H., Sands, D., Sudbrock, H.: Assumptions and guarantees for compositional noninterference. In: Proc. of Computer Security Foundations Symposium (CSF). pp. 218–232. Cernay-la-Ville, France (Jun 2011)
30. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Proc. of Verification, Model Checking, and Abstract Interpretation (VMCAI). pp. 41–62. Springer (2016)
31. Murray, T.: Short paper: On high-assurance information-flow-secure programming languages. In: Proc. of Workshop on Programming Languages and Analysis for Security (PLAS). pp. 43–48 (2015)
32. Murray, T., Matichuk, D., Brassil, M., Gammie, P., Bourke, T., Seefried, S., Lewis, C., Gao, X., Klein, G.: seL4: from general purpose to a proof of information flow enforcement. In: Proc. of Symposium on Security and Privacy (S&P). pp. 415–429. San Francisco, CA (May 2013)
33. Murray, T., Sabelfeld, A., Bauer, L.: Special issue on verified information flow security. Journal of Computer Security **25**(4-5), 319–321 (2017)
34. Murray, T., Sison, R., Engelhardt, K.: COVERN: A logic for compositional verification of information flow control. In: Proc. of European Symposium on Security and Privacy (EuroS&P). London, United Kingdom (Apr 2018)
35. Murray, T., Sison, R., Pierzchalski, E., Rizkallah, C.: Compositional verification and refinement of concurrent value-dependent noninterference. In: Proc. of Computer Security Foundations Symposium (CSF). pp. 417–431 (Jun 2016)

36. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services uses formal methods. Communications of the ACM **58**(4), 66–73 (2015)
37. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283 (2002)
38. O'Hearn, P.W.: Resources, concurrency and local reasoning. In: International Conference on Concurrency Theory (CONCUR). pp. 49–67. Springer (2004)
39. O'Hearn, P.W.: Continuous reasoning: Scaling the impact of formal methods. In: Proc. of Logic in Computer Science (LICS). pp. 13–25. ACM (2018)
40. O'Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. ACM Transactions on Programming Languages and Systems (TOPLAS) **31**(3), 11 (2009)
41. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: Proc. of Computer Aided Verification (CAV). pp. 773–789. Springer (2013)
42. Polikarpova, N., Sergey, I.: Structuring the synthesis of heap-manipulating programs. Proceedings of the ACM on Programming Languages **3**(POPL), 72 (2019)
43. Prabawa, A., Al Ameen, M.F., Lee, B., Chin, W.N.: A logical system for modular information flow verification. In: Proc. of Verification, Model Checking, and Abstract Interpretation (VMCAI). pp. 430–451. Springer (2018)
44. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. of Logic in Computer Science (LICS). pp. 55–74. IEEE (2002)
45. Sabelfeld, A., Sands, D.: Probabilistic noninterference for multi-threaded programs. In: Proc. of Computer Security Foundations Workshop (CSFW). pp. 200–214. IEEE (2000)
46. Vafeiadis, V.: Concurrent separation logic and operational semantics. In: Proc. of Mathematical Foundations of Programming Semantics (MFPS). pp. 335–351 (2011)
47. Vafeiadis, V., Narayan, C.: Relaxed separation logic: A program logic for C11 concurrency. In: Proc. of Object Oriented Programming Systems Languages & Applications (OOPSLA). pp. 867–884. ACM (2013)
48. Volpano, D., Smith, G.: Probabilistic noninterference in a concurrent language. Journal of Computer Security **7**(2,3), 231–253 (1999)
49. Yang, H.: Relational separation logic. Theoretical Computer Science **375**(1-3), 308–334 (2007)
50. Zheng, L., Myers, A.C.: Dynamic security labels and static information flow control. International Journal of Information Security **6**(2–3) (Mar 2007)

A Command Semantics

Symmetric parallel rules in which c_2 is scheduled under the action $\langle 2 \rangle$ omitted.

$$\frac{s' = s(x \mapsto [\![e]\!]_s)}{(\mathbf{run}\ x := e, L, s, h) \xrightarrow{\langle \tau \rangle} (\mathbf{stop}\ L, s', h)} \qquad \frac{[\![e]\!]_s \notin \mathrm{dom}\ (h)}{(\mathbf{run}\ x := [e], L, s, h) \xrightarrow{\langle \tau \rangle} \mathbf{abort}}$$

$$\frac{[\![e]\!]_s \in \mathrm{dom}\ (h) \qquad s' = s(x \mapsto h([\![e]\!]_s))}{(\mathbf{run}\ x := [e], L, s, h) \xrightarrow{\langle \tau \rangle} (\mathbf{stop}\ L, s', h)} \qquad \frac{[\![e_1]\!]_s \notin \mathrm{dom}\ (h)}{(\mathbf{run}\ [e_1] := e_2, L, s, h) \xrightarrow{\langle \tau \rangle} \mathbf{abort}}$$

$$\frac{[\![e_1]\!]_s \in \mathrm{dom}\ (h) \qquad h' = h([\![e_1]\!]_s \mapsto [\![e_2]\!]_s)}{(\mathbf{run}\ [e_1] := e_2, L, s, h) \xrightarrow{\langle \tau \rangle} (\mathbf{stop}\ L, s, h')}$$

$$\frac{l \in L \qquad L' = L \setminus \{l\}}{(\mathbf{run}\ \texttt{lock}\ l, L, s, h) \xrightarrow{\langle \tau \rangle} (\mathbf{stop}\ L', s, h)}$$

$$\frac{l \notin L \qquad L' = L \cup \{l\}}{(\mathbf{run}\ \texttt{unlock}\ l, L, s, h) \xrightarrow{\langle \tau \rangle} (\mathbf{stop}\ L', s, h)} \qquad \frac{(\mathbf{run}\ c_1, L, s, h) \xrightarrow{\sigma} \mathbf{abort}}{(\mathbf{run}\ c_1; c_2, L, s, h) \xrightarrow{\sigma} \mathbf{abort}}$$

$$\frac{(\mathbf{run}\ c_1, L, s, h) \xrightarrow{\sigma} \mathbf{abort}}{(\mathbf{run}\ c_1 \parallel c_2, L, s, h) \xrightarrow{\langle 1 \rangle \cdot \sigma} \mathbf{abort}} \qquad \frac{(\mathbf{run}\ c_1, L, s, h) \xrightarrow{\sigma} (\mathbf{stop}\ L', s', h')}{(\mathbf{run}\ c_1; c_2, L, s, h) \xrightarrow{\sigma} (\mathbf{run}\ c_2, L', s', h')}$$

$$\frac{(\mathbf{run}\ c_1, L, s, h) \xrightarrow{\sigma} (\mathbf{run}\ c_1', L', s', h')}{(\mathbf{run}\ c_1; c_2, L, s, h) \xrightarrow{\sigma} (\mathbf{run}\ c_1'; c_2, L', s', h')}$$

$$\frac{(\mathbf{run}\ c_1, L, s, h) \xrightarrow{\sigma} (\mathbf{stop}\ L', s', h')}{(\mathbf{run}\ c_1 \parallel c_2, L, s, h) \xrightarrow{\langle 1 \rangle \cdot \sigma} (\mathbf{run}\ c_2, L', s', h')}$$

$$\frac{(\mathbf{run}\ c_1, L, s, h) \xrightarrow{\sigma} (\mathbf{run}\ c_1', L', s', h')}{(\mathbf{run}\ c_1 \parallel c_2, L, s, h) \xrightarrow{\langle 1 \rangle \cdot \sigma} (\mathbf{run}\ c_1' \parallel c_2, L', s', h')}$$

$$\frac{\text{if } s \models b \text{ then } c' = c_1 \text{ else } c' = c_2}{(\mathbf{run}\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, L, s, h) \xrightarrow{\langle \tau \rangle} (\mathbf{run}\ c', L, s, h)}$$

$$\frac{s \not\models b}{(\mathbf{run}\ \texttt{while}\ b\ \texttt{do}\ c, L, s, h) \xrightarrow{\langle \tau \rangle} (\mathbf{stop}\ L, s, h)}$$

$$\frac{s \models b}{(\mathbf{run}\ \underbrace{\texttt{while}\ b\ \texttt{do}\ c}_{\omega}, L, s, h) \xrightarrow{\langle \tau \rangle} (\mathbf{run}\ (c; \omega), L, s, h)}$$

$$k \xrightarrow{\langle\,\rangle}{}^{*}\ k \qquad \frac{k \xrightarrow{\sigma_1} k' \qquad k' \xrightarrow{\sigma_2}{}^{*}\ k''}{k \xrightarrow{\sigma_1 \cdot \sigma_2}{}^{*}\ k''}$$

B Proofs

Proof of Lemma 1

If $(s,h),(s',h') \models_\ell P$, then $h \overset{A}{\equiv} h'$ for $A = \mathrm{lows}_\ell(P,s)$.

Proof. By induction on the structure of P, noting that $\mathrm{lows}_\ell(\_,s)$ contains locations of the corresponding sub-heap only. □

Proof of Lemma 2

$\ell \vdash \{P\}\ c\ \{Q\}$ implies $\mathrm{secure}^n_\ell(P,c,Q)$ for every $n \geq 0$.

Proof (Outline). By induction on the derivation of the validity of the judgement. Noting that $n = 0$ is trivial, we may unfold the recursion of the security definition once to prove the base cases of assignment, load, store, and locking, which then follow from the respective side conditions of the proof rules.

For rules If and While, the side condition $b :: \ell$ guarantees that the test evaluates equivalently in the two states and thus execution proceeds with the same remainder program.

Except for If, all remaining rules need a second induction on n to stepwise match security of the premise to security of the conclusion (e.g. over the steps of the first command in a sequential composition $c_1; c_2$).

The rule Frame instantiates the frame F with the same assertion in each step, whereas Par uses the frame F to preserve the current precondition P_2 of c_2 over steps of c_1 and vice-versa. □

Proof of Corollary 1

Given a command c and initial states $(s_1,h_1),(s_1',h_1') \models_\ell P \star \mathrm{invs}(L_1)$ and two executions under the same schedule to resulting configurations k and k' respectively, then $\ell \vdash \{P\}\ c\ \{Q\}$ implies $k \neq \mathbf{abort} \wedge k' \neq \mathbf{abort}$.

Proof. By induction on the number of steps n of the executions from $\mathrm{secure}^n_\ell(P,c,Q)$ via Lemma 2. □

Proof of Theorem 1

Given a command c and initial states $(s_1, h_1), (s_1', h_1') \models_\ell P \star \text{invs}(L_1)$ and two complete executions under the same schedule σ

$$(\textbf{run}\ c, L_1, s_1, h_1) \xrightarrow{\sigma}{}^{*} (\textbf{stop}\ L_2, s_2, h_2)$$
$$(\textbf{run}\ c_i, L_i, s_i', h_i') \xrightarrow{\sigma}{}^{*} (\textbf{stop}\ L_2, s_2', h_2')$$

then $\ell \vdash \{P\}\ c\ \{Q\}$ implies $(s_2, h_2), (s_2', h_2') \models_\ell Q \star \text{invs}(L_2)$.

Proof. By induction on the number of steps n of the executions from $\text{secure}_\ell^n(P, c, Q)$ via Lemma 2. □

Proof of Theorem 2

Given a command c, and initial states $(s_1, h_1), (s_1', h_1') \models_\ell P \star \text{invs}(L_1)$ then $\ell \vdash \{P\}\ c\ \{Q\}$ implies $h_i \overset{A}{\equiv} h_i'$, where $A = \text{lows}_\ell(P, s_1)$, for all pairs of heaps h_i and h_i' arising from executing the same schedule from each initial state.

Proof. By induction on the number of steps i up to that state from $\text{secure}_\ell^i(P, c, Q)$ via Lemma 2 we have $\text{lows}_\ell(P \star \text{invs}(L_1), s_1) \subseteq \text{lows}_\ell(P_i \star \text{invs}(L_1), s_i)$ transitively over the prefix, where P_i and s_i are from the i-th state. The theorem then follows from Lemma 1 in Section 3.1. □