Assume but Verify: Deductive Verification of Leaked Information in Concurrent Applications (Extended Version)

Toby Murray
toby.murray@unimelb.edu.au
University of Melbourne
Australia

Mukesh Tiwari*
mt883@cam.ac.uk
University of Cambridge
United Kingdom

Gidon Ernst
gidon.ernst@lmu.de
LMU Munich
Germany

David A. Naumann
naumann@cs.stevens.edu
Stevens Institute of Technology
USA

ABSTRACT

We consider the problem of specifying and proving the security of non-trivial, concurrent programs that intentionally leak information. We present a method that decomposes the problem into (a) proving that the program only leaks information it has declassified via **assume** annotations already widely used in deductive program verification; and (b) auditing the declassifications against a declarative security policy. We show how condition (a) can be enforced by an extension of the existing program logic SecCSL, and how (b) can be checked by proving a set of simple entailments. Part of the challenge is to define respective semantic soundness criteria and to formally connect these to the logic rules and policy audit. We support our methodology in an auto-active program verifier, which we apply to verify the implementations of various case study programs against a range of declassification policies.

1 INTRODUCTION

Methods for proving that programs are not just functionally correct, but also maintain confidential information *securely*, have been applied to realistic software like operating system kernels [29, 59], encompassing features like concurrency [46, 60] and pointers [28, 39]. These methods have also been embodied in auto-active program verification tools like SecC [37] and a variant of Viper [35, 58]. In the *auto-active* verification paradigm, popularized by tools like Dafny [50, 51], VeriFast [44], and Why3 [38], programs are verified semi-automatically via annotations added to their source code, supporting a high degree practical usability (made possible by advances in automated backend provers like SMT solvers).

The standard criterion for security is noninterference [40] which guarantees absence of information leaks. However, as has been noted repeatedly [5, 17, 18, 34, 71, 72, 77, 81], practical programs that handle sensitive information almost always intentionally reveal some part of that information. Such an act of *declassification* is deemed secure if it adheres to a given high-level policy. As an example (Section 2), we may release statistics like the average of numbers in a data set if this set is sufficiently large and/or homogeneous enough such that the leaked information about the individual data points is acceptably low. One approach to reasoning about declassification relies on relational *assume statements* [9, 25], and this is used in tools like SecC [37] and a variant of Viper [58] —but these tools make no formal connection with high level policy.

A high level policy designates security levels for input and output channels, with a basic interpretation that observations at a given level should reveal no information about inputs except those at or below the given level – and except for designated intentional releases. Such a declarative policy designates *what* information may be released to observers at lower level, and *when*, i.e., under what conditions. The conditions refer to observable data values and events, like the size of the data set in our example. The precise meaning of a declarative policy can be formalized in terms of observer *knowledge* [5, 7–9, 18].

In a nutshell, this paper contributes an auto-active verification tool that provably enforces declarative high level policies for concurrent C code, and its evaluation through challenging case studies. Security is defined in terms of knowledge and proved using a relational logic. Previously, this approach has only been sketched [26, Section VII, B] for a much simpler program semantics and with no implemented tool.

The **first contribution** of this paper is to formalize the security property given by a high level policy, for a programming model with concurrent threads and dynamically allocated mutable objects, with respect to a standard threat model. That is, the property makes the strong guarantee of *constant-time* security [2], which precludes secret-dependent branching and loads/stores for memory addresses that are secret-dependent. Thus, even in the presence of variable latency induced by instruction- and data-caches, a program's running time is independent of secrets. This is more restrictive than some security conditions in the literature, as discussed in Section 8, but the threat model is well suited to many context where C programs are used.

Our formalization disentangles a *policy-agnostic* [80] security property from the *policy-specific* property associated with a high-level declarative policy. This supports an important methodological point. In a well designed program, declassifications occur only at particular places in the code. To verify a program, we designate these places by assume statements. The policy-agnostic property says that no releases occur —i.e., the observer never learns anything— except at execution steps with assume statements. The policy-specific property says that those steps only release what is allowed by the policy, and only when the associated release condition holds.

*This work was carried out while the author was at the University of Melbourne.

Our **second contribution** is a deductive proof system which supports reasoning about assume and assert statements. A high level policy specifies conditions under which particular values may be released. Because the policy is program-independent, it expresses release conditions as predicates on the program's I/O history (inspired by [9, 72]). To verify a program with respect to a high level policy, each assumption should be justified by an assertion of a condition in the high level policy that licenses the release —we call this *policy audit*. The assertion makes use of a ghost variable that records the history. We prove that (a) the proof system is sound wrt. the policy-agnostic property, and (b) if the program passes the policy audit then it satisfies the policy-specific security property. The proofs are mechanized in Isabelle/HOL.

We choose to base the second contribution on the existing logic SecCSL [37], as it comes not only with a tool implementation but also a mechanized soundness proof that SecCSL satisfies strong noninterference (absent assume statements). We extend this framework by uniformly capturing security-relevant semantic actions of the program (assumptions, actions/outputs, memory access) as the basis of the policy-agnostic security guarantee. In doing so, we inherit from SecCSL its capabilities for proving security of concurrent programs with lock-based synchronization.

The **third contribution** of this work is a practical demonstration of the approach. We have implemented the approach in the auto-active verifier Verdeca, including support for declassification and policy audit. Verdeca treats a subset of C, and is targeted at verifying concurrent shared-memory programs that use lock-based synchronization. We leave verification of lock-free programs, exposed to weak-memory effects [79], to future work.

We carry out several challenging case studies: The first, a location service for mobility traces [21] leverages domain knowledge about privacy budgets wrt. adding *planar laplacian noise* [4] to achieve differential privacy [33] via a suitable policy; verifying that this budget is never exceeded. The second case study, a sealed-bid auction server ensures that no client learns anything about the current maximum bid until the auction closes. Each client's TCP connection is serviced by a separate thread to ensure that no client can block the server's progress and thereby game the auction. With a variant of this example we demonstrate furthermore *policy composition*. The third case study is a verified constant-time implementation of the popular game Wordle, where rules of the game induce an interesting value-dependent, multi-level declassification policy with each move of the player, i.e., revealing information about the characters and positions guessed correctly to that player, but not to other concurrent players nor to the general public. The final case study considers secure, constant-time, private learning, specifically differentially private gradient descent to infer a simple linear model, designed for deployment in a federated learning scenario [78].

Section 2 gives a high-level overview of the ideas. The threat model is made explicit in Section 3. The semantic and logical foundations are presented in Section 4. The policy-agnostic and policy-specific guarantees are formalized in Section 5 and Section 6, respectively. Case-studies are presented in Section 7, and Section 8 compares to related work and concludes. Appendix B sketches proofs for our main results, which are mechanised in Isabelle/HOL.

```
struct avg_state { int count; int sum; };

struct avg_state * avg_lock();
void avg_unlock(struct avg_state *st);

int avg_get_input();
  _(ensures result :: high)
  _(requires 𝓗(tr))
  _(ensures  𝓗(tr · result))

void print_average(int value);
  _(requires value :: low)

void avg_sum_thread() {
  while(true) {
    struct avg_state * st = avg_lock();
    int i = avg_get_input();
    st->count += 1;
    st->sum += i;
    avg_unlock(st);
  }
}

void avg_declass_thread() {
  struct avg_state * st = avg_lock();
  if (st->count >= 6) {
    int avg = st->sum / st->count;
    _(assume avg :: low)                                    (†)
    print_average(avg);
  }
  avg_unlock(st);
}
```

Figure 1: Declassifying the average of at least 6 inputs.

2 MOTIVATION AND OVERVIEW

Consider the program in Fig. 1, inspired by the *running average* example from Schoepe et al. [72]. It represents the state-of-the-art in terms of the size of concurrent programs that have been verified for secure declassification prior to Verdeca; our case studies in Section 7 are an order of magnitude larger.

Here, two threads cooperate to compute and declassify an aggregate statistic (in this case a simple average) calculated over purportedly sensitive inputs. This program's declarative *security policy* allows the average of the inputs received to be declassified so long as that average has been calculated over at least 6 inputs.

One thread repeatedly waits for new inputs to arrive and computes a running sum as well as the count of the inputs received so far; the second thread reads these values from the shared state, and prints out the average but only if at least 6 inputs have been counted in the shared state, to honor the security policy.

This program makes use of four *external library* functions (those in Fig. 1 without an implementation). Such functions are part of the application's trusted computing base (TCB) and are trusted to be correct and secure. The external functions `avg_lock()` and

`avg_unlock()` implement a mutex that is used to coordinate access to the shared state (running input count and sum) between the two threads. Function `avg_get_input()` gets the next input of high sensitivity, i.e., it returns a secret value. This is specified in terms of its postcondition inside the annotation `_(ensures result:: high)`, where $e :: \ell$ denotes that the expression e holds a value that conforms to classification by security label ℓ. The other two annotations mentioning $\mathcal{H}$ track the input/output history of the program as a trace tr of events that are relevant to the declassification policy, here the trace is extended by the result of the function. Function `print_average()` prints out its argument to a public channel, its contract similarly mentions a classification, this time, the precondition inside `_(requires value :: low)` specifies that any argument passed to this function should be of low sensitivity, i.e., public.

What does it mean for this program to be secure? Clearly, the program does not satisfy noninterference [40], which requires that the attacker *never* learns secret values: since the values stored in `count` and `sum` have been computed from the secret inputs and the attacker will in general learn something from the call to `print_average()`. Indeed, proving the program secure with vanilla SecCSL [37] will not be possible, therefore we add **assume** statements in Section 4.2, which make explicit the decision that the value of `avg` can now be considered to be public right before the call to `print_average()`. Fig. 1 shows use of an assume statement to make explicit the declassification (†).

From the point of a verification engineer, **assume** statements express that from this point onwards in the program's execution, the attacker is assumed to know the declassified value [26], as with `avg` in the example above. And indeed, merely adding assume statements to the logic is easy, and this is how they have been used in other approaches [35]. But what justifies such an assumption? Just as with assuming the absence of hash collisions in cryptographic applications [32], the act of *declassification should be justified with respect to high level policy.* The practical challenges in verifying the program are fundamentally on a different level of abstraction than the concerns related to formulating and validating declassification policies. Therefore, we argue, security in the presence of declassification policies intrinsically suggests separating two concerns: (a) the code leaks no information except as made explicit in assumptions, and (b) all assumptions are justified by the high level policy. For both concerns individually we provide a security property and a sound verification method.

> **Goal: policy-agnostic security guarantee.** Information leaks in a *verified* program can always be traced back to a prior failed assumption.

To make this intuition precise we consider the attacker's knowledge at a given point in an execution (called the "major run" [13]), in terms of their *uncertainty* [7, 26, 72] about the initial secrets. The uncertainty is the set of runs that are consistent with what the attacker is able to observe about the major run. A given "minor run" gets removed from the uncertainty at any step of the major run where the attacker can observe something inconsistent with that minor run. Our formalization is based on the notion of a *schedule*, a generic semantic model that relates such observations to points in control flow annotated with explicit assumptions about attacker

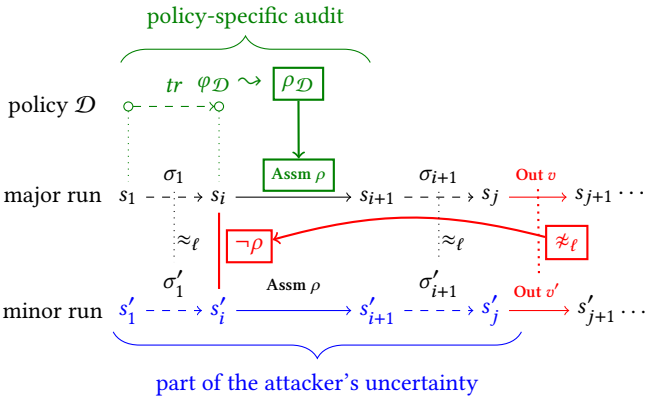

Figure 2: Visual representation that relates a major run to a hypothetical minor run, where the mismatch in outputs can be traced to an earlier failed assumption.

knowledge. In Fig. 2 the attacker is uncertain about the possible initial state s'_1 because there exists a minor run up to the state s'_j whose schedule (primed σ') is observationally equivalent, written $\approx_\ell$, to the schedule of the major run (unprimed σ). In the j-th step, an information leak occurs when the outputs mismatch $v \neq v'$. The policy-agnostic security guarantee ensures that this mismatch can always be explained by an assumption ρ, occurring at an earlier i-th step, that is not satisfied at that point (cf. red backwards arrow). Dually, the attacker is not allowed to exclude runs from their uncertainty that have no such assumption violation. We formally prove this as Theorem 5.6 in Section 5.

Note that formulas ρ like $e :: \ell$ are interpreted relationally over pairs of states, specifically $e :: \mathbf{low}$ means that e evaluates to the same value in both. In the example from Fig. 1, therefore, the assumption $\rho \triangleq$ `avg` $::$ **low** (marked by (†)) must have failed between some prior s_i and s'_i, such that the difference between v and v' is *caused* by the different values of `avg`.

A declassification policy $\mathcal{D}(tr)$ is formulated over a representation of the *execution trace* tr, that is collected as auxiliary "ghost" state in the verification as a syntactic expression of a suitable sequence data type. The trace is tracked with an abstract I/O predicate $\mathcal{H}(tr)$ that specifies that the trace denoted by expression tr is the current history of the program at that point and that tr is well-formed wrt. the application (examples in Section 7).

A policy $\mathcal{D}(tr)$ over trace tr takes the form $\varphi_{\mathcal{D}}(tr) \rightsquigarrow \rho_{\mathcal{D}}(tr)$. It consists of a *condition* $\varphi_{\mathcal{D}}$ on traces which states *when* a declassification is permitted, and a relational *release formula* $\rho_{\mathcal{D}}$, which encodes *what* information is allowed to be released. For the example, the policy requires that the trace tr has a length of at least 6; it then justifies to classify the average over the numbers stored in tr as **low**:

$$\mathcal{D}(tr) \quad \triangleq \quad \mathsf{length}(tr) \geq 6 \rightsquigarrow \mathsf{sum}(tr)/\mathsf{length}(tr) :: \mathbf{low}$$

Here, the critical issue is to enforce whether all assumptions made in the program are covered by the policy, with respect to the symbolic path constraints at that point, which we call an *audit* (exemplified below).

> **Goal: certified adherence to policy.** In a program that has been correctly *audited* with respect to a declassification policy, the information leak associated with each assumption is bounded by the policy.

In Fig. 2 this is depicted at the top (green): each possibly failing assumption has to be justified from the policy $\mathcal{D}$ with respect to the trace prefix *tr* at that point. Together with the policy-agnostic guarantee, this implies that from the execution of a verified and audited program an attacker can only gain knowledge that is allowed by the policy. We formalize this as Theorem 6.5 in Section 6 by integrating a declarative semantics of policies over traces with the program's schedules, i.e., with the ground-truth about the execution.

To prove the audit (i.e. that each assumption that introduces a leak is justified by the declassification policy $\mathcal{D}$) we make use of *invariants* established by the verification that connect the program's state to the abstract trace. For the example, this means that the verification attaches the following resource invariant to the shared state (acquired when calling `avg_lock()` and required to be true when calling `avg_unlock()`):

$$\mathsf{inv}(tr, \mathtt{st}) \;\hat{=}\; \mathcal{H}(tr) \wedge \mathtt{st\text{->}count} = \mathsf{length}(tr) \wedge \mathtt{st\text{->}sum} = \mathsf{sum}(tr) \tag{1}$$

Together with path condition `st->count >= 6` from the `if` test, the audit formally certifies that right before the assume, the policy condition $\varphi_{\mathcal{D}}(tr)$ is entailed. Auditing then requires us to prove that the policy release formula $\rho_{\mathcal{D}}(tr)$ (what the policy allows to be declassified) implies the assumption `avg` ::**low** (what the program has actually declassified), which holds under the resource invariant and thus this example satisfies secure declassification. The full proof for this example appears in Appendix A.

In summary, successful verification of an annotated program ensures that every increase in attacker knowledge occurs following an assumption that allows the new knowledge. Successful policy audit ensures that for any assumption there is a policy clause $\varphi_{\mathcal{D}}(tr) \rightsquigarrow \rho_{\mathcal{D}}(tr)$ such that the path condition P at the assumption implies the release condition $\varphi_{\mathcal{D}}(tr)$ which in turn validates $\rho_{\mathcal{D}}(tr)$, i.e., the released information is allowed by the policy.

3 THREAT MODEL

We assume a program being verified that contains a number of top-level functions like `avg_declass_thread()` and `avg_sum_thread()` in Fig. 1. These top-level functions are necessarily invoked from unverified, trusted *wrapper* code like `main()` that may invoke multiple instances of each function and in parallel. Verified application functions also make use of external library functions like `print_average()` that are not to be verified and must be trusted.

Our approach rests on a number of assumptions, including [61] *adversary expectations*—those assumptions that apply to the attacker—and *domain hypotheses*—those that apply to the program's environment. We also make various meta-assumptions that apply to the verification approach itself.

Adversary Expectations. We assume a passive attacker with arbitrary security level ℓ who knows the verified program's source code and its proof (as expressed in the annotations in the source). We assume that the attacker can observe all data passed to any external library function whose precondition requires that data to be ℓ or below (including **low**). Similarly, we assume that the attacker initially knows only those program data that have been either marked at ℓ or below in the precondition of a top-level application function, or in the postcondition of an external library function. Without loss of generality, we assume that all secrets are contained somewhere in the initial program state (e.g. via a standard, deterministic oracle semantics [27, 59] for functions like `avg_get_input()`, not elaborated here). Following the *constant-time* security threat model [2], we assume the attacker can observe not only the program's execution time but also its memory access pattern and which conditional branches it takes. These latter are made observable via timing effects induced by microarchitectural elements like caches. The attacker can also observe the program's concurrent schedule [37, 39].

Domain Hypotheses. We assume that the unverified code is correct and secure: any preconditions of top-level verified application functions are always satisfied whenever they are invoked by non-verified code, and all external library functions satisfy their contracts. We assume that the verified program executes faithfully atop an operating system whose scheduler does not leak sensitive information [59] and is insulated from transient execution effects [19].

Meta Assumptions. Verdeca implements the logic presented in this paper (Section 4 with the extensions from Section 6), not for the simple command language defined in Section 4 over which our soundness theorems are proved, but for a substantial fragment of the C language. We assume that this implementation is faithful to the theoretical ideas of this paper as well as to the semantics of the supported part of C (cf. Section 7). A small caveat is that Verdeca currently models signed integers as mathematical integers, so we assume that traditional verification methods have also been applied to ensure absence of overflow, which is an orthogonal problem to those considered by this work. Since Verdeca relies on Z3 to discharge verification conditions, we assume that Z3 has no soundness bugs that affect our proofs. These assumptions are common to auto-active verifiers. Similarly, for the mechanized proofs, we rely on the soundness of Isabelle/HOL, which thanks to its small-kernel architecture is highly trustworthy.

Claim. Then we claim that if verified with Verdeca, the program in question will adhere to its security policy as specified, against the aforementioned attacker. Specifically this attacker can learn no information other than what the program declassifies, and all declassifications are in accordance with the security policy.

4 BACKGROUND: SECCSL

Security Concurrent Separation Logic (SecCSL) is a program logic proposed in [37] for proving timing-sensitive noninterference of concurrent programs. It extends Concurrent Separation Logic [62] by adding new assertions to capture security-related properties and by adapting the existing proof rules to ensure that these are maintained soundly. We adapt and extend SecCSL as the foundation of Verdeca, for its native support for compositional, modular, implementation-level reasoning.

Judgements in the logic have the form

$$\vdash_{\ell} \{P\}\ c\ \{Q\}$$

for a command c and pre-/postcondition P and Q, where ℓ denotes the level of the attacker, typically **low**. It implies semantically that if the program c is executed when precondition P holds then, c's execution will be memory-safe and when c terminates the postcondition Q will hold (partial correctness). Moreover, with respect to the adversary assumptions stated in Section 3, the execution will not leak information to the ℓ-level attacker. This includes via *timing channels* as the proof rules enforce that programs never branch on secrets nor perform secret-dependent memory accesses.

In this section we present the necessary background on the assertion language, the proof rules, and the semantic foundations. We rely on these preliminaries in Section 5 and Section 6, where we will discuss how the guarantees entailed by $\vdash_\ell \{P\}\ c\ \{Q\}$ are formalized—these guarantees are the key difference between our work and SecCSL.

The logic is defined for a core programming language with commands c:

$$\begin{aligned} c \ ::= \ & x := e \mid \textbf{lock}\ l \mid \textbf{unlock}\ l \mid c_1; c_2 \mid c_1 \parallel c_2 \mid \\ & \textbf{if}\ \phi\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ \phi\ \textbf{do}\ c \mid \\ & \textbf{assume}\ \rho \mid \textbf{output}\ \ell'\ e_v \mid x := [e_p] \mid [e_p] := e_v \mid \quad (\dagger) \\ & \textbf{trace}\ e \quad (\ddagger) \end{aligned}$$

The details of most commands present in SecCSL already remain unchanged: this includes assignments, locking, sequential/parallel composition, conditionals, and while loops.

Extensions to SecCSL: Outputs, assumptions, and **trace** e are new wrt. SecCSL. The two commands for memory load and store are modeled differently here to capture constant-time security. The commands on the third line (†) are relevant to the program's policy-agnostic security guarantee as discussed in Section 5. The command **trace** e (‡) represents the occurrence of application-specific events with data e, such as calls to `avg_get_input()` where e captures its return value. These events are not visible to the attacker but instead give a declarative account of the guarantees entailed by policy enforcement, as explained in Section 6.

Notation: For a schedule σ, formalized as a mathematical sequence of actions (defined later), we write $|\sigma|$ to denote its the length, $\sigma_1 \cdot \sigma_2$ denotes concatenation, $\langle a \rangle$ is the singleton sequence with element a, σ_i is the i–th entry if $i < |\sigma|$ and $\sigma_{|i}$ is the prefix of length i. Program states make use of stores s, modeled as mathematical maps; we write $s(x)$ for lookup and $s(x := v)$ for map override. Heaps h in addition have a domain $\mathrm{dom}(h)$, and we write $h = [a \mapsto v]$ for a singleton heap that maps address a to value v and $h_1 \uplus h_2$ for the union of two heaps, implying that they need to have disjoint domains.

4.1 Expressions and Assertions

The typed language of expressions e includes boolean formulas $\phi : \mathit{Bool}$ and a designated sort *Label* for security labels, including at least the constants **low** and **high** and a binary relation $\sqsubseteq$ that satisfies the lattice axioms. The assertion language to formulate pre-/postconditions includes the standard connectives, quantifiers, and separation logic primitives points-to and separating conjunction:

$$\begin{aligned} &\text{assertion } P \ ::= \\ &\quad \phi \mid e :: \ell' \mid \textbf{emp} \mid e_p \mapsto e_v \mid P_1 \star P_2 \mid P_1 \implies P_2 \mid \exists x.\, P \mid \cdots \end{aligned}$$

$$\begin{aligned} (s,h),(s',h') \models_\ell \phi \ &\triangleq\ s, s' \models_\ell \phi \wedge h = h' = \varnothing \quad \text{for } \phi \text{ pure} \\ \text{where} \quad s, s' \models_\ell \phi \ &\triangleq\ [\![\phi]\!]_s \wedge [\![\phi]\!]_{s'} \\ s, s' \models_\ell e :: \ell' \ &\triangleq\ [\![\ell']\!]_s \sqsubseteq \ell \wedge [\![\ell']\!]_{s'} \sqsubseteq \ell \implies [\![e]\!]_s = [\![e]\!]_{s'} \end{aligned}$$

$$\begin{aligned} (s,h),(s',h') \models_\ell e_p \mapsto e_v \ &\triangleq\ h = [[\![e_p]\!]_s \mapsto [\![e_v]\!]_s] \\ &\quad \text{and } h' = [[\![e_p]\!]_{s'} \mapsto [\![e_v]\!]_{s'}] \\ (s,h),(s',h') \models_\ell P_1 \star P_2 \ &\triangleq\ h = h_1 \uplus h_2 \text{ and } h' = h_1' \uplus h_2' \\ &\quad \text{and } (s,h_i),(s',h_i') \models_\ell P_i \text{ for } i = 1,2 \\ (s,h),(s',h') \models_\ell P_1 \implies P_2 \ &\triangleq\ (s,h),(s',h') \models_\ell P_1 \\ &\quad \text{implies } (s,h),(s',h') \models_\ell P_2 \\ (s,h),(s',h') \models_\ell \exists x.\, P \ &\triangleq\ \text{there are } v, v' \text{ with} \\ &\quad (s(x := v), h), (s'(x := v'), h') \models_\ell P \end{aligned}$$

Figure 3: Relational semantics of assertions.

In addition, we have *value sensitivity* or classification $e :: \ell'$, which expresses that the value of e is safe to be known for an ℓ' attacker, where $e :: \textbf{low}$ is the strongest such assertion and $e :: \textbf{high}$ is just true. Expressive power comes from the reflection of the security lattice into the assertion language, such that labels ℓ are symbolic expressions, too. For example, $e :: (d\ ?\ \textbf{high} : \textbf{low})$ denotes a classification of e conditional on the current value of d, where $(\_\ ?\ \_ : \_)$ is an if-then-else expression. Spatial assertions include the empty heap **emp**, the points-to predicate $e_p \mapsto e_v$, i.e., e_p points to valid memory containing value e_v, and the separating conjunction $P_1 \star P_2$, i.e., assertions P_1 and P_2 hold on disjoint parts of the heap, respectively, as usual [62, 68].

An assertion is called *pure* if it does not make reference to the heap, and it is called *relational* if it includes a classification. We denote pure relational formulas by letter ρ in the following.

Semantically, expressions e are evaluated over a store s, written $[\![e]\!]_s$ in the standard way. Assertions, in contrast, are evaluated over a *pair of states*, each consisting of a store s and a heap h, with respect to the attacker level ℓ. We write $s, s' \models_\ell \rho$ when pure assertion ρ holds and $(s,h),(s',h') \models_\ell P$ that spatial assertion P holds, where intuitively s and h are taken from the actual "major" run of the program that is compared to s' and h' from some hypothetical "minor" run in Beringer's terminology [13] (cf. Fig. 1).

The semantics of assertions is shown in Fig. 3. Non-relational formulas ϕ must hold in both states individually. Value sensitivity $e :: \ell'$ enforces agreement of the values $[\![e]\!]_s$ and $[\![e]\!]_{s'}$ in both states for an attacker at level ℓ that is at least as high as ℓ'. Note that this semantics improves on original SecCSL [37] by not requiring ℓ' to agree, which is crucial for specifying the Wordle security policy (Section 7.3). As in SecCSL [37], pure assertions impose an empty heap. Spatial assertions extend standard Separation Logic (SL) semantics to pairs of states point-wise.

In contrast to the grammar shown, the original SecCSL supports in addition *memory location sensitivity*, written $e_p \overset{\ell'}{\mapsto} e_v$, which

expresses that both memory *access* via address e_p is observable to an ℓ'-attacker as well as the actual value stored therein. However, this feature comes with some trade-offs, e.g. assertions are restricted to the positive fragment of SL and logical entailment becomes somewhat complex. In Section 4.2 below we offer a different approach based on the standard points-to assertion that avoids these limitations.

4.2 Proof Rules

The proof rules of SecCSL to derive judgements $\vdash_\ell \{P\}\ c\ \{Q\}$ are all "matched" (aka "synchronous") rules, where the control flow of program c is always the same between the major and minor run. The approach is adequate for *timing-sensitive* security as discussed in Section 3 and integrates nicely into existing logics (though it is not adequate for weaker security properties that allow some branching on secrets). Appendix A shows the proof sketch built from these rules for the example from Section 2.

The proof rules of SecCSL work exactly like those of traditional Concurrent SL [62, 68] except that the rules for **if** and **while** in Fig. 4 enforce that the respective branch condition ϕ is not secret from the view of the ℓ-attacker. A similar requirement applies to *logical* case splits (rule Split): semantically, there are four combinations how ϕ could evaluate, of which our assertion language can represent two (cf. second line in Fig. 3). These side conditions related to security are highlighted in light gray. Rules for sequential and parallel composition, acquiring and releasing locks, as well as the frame and consequence rules are entirely standard, see e.g. [68] (sequential fragment) and [41, Fig. 2] (concurrency and locks). The locking rules transfer ownership of a *lock invariant* denoted $\mathsf{inv}(l)$ for lock l such as the one in Eq. (1).

The proof rules for atomic commands except **trace** e are shown in Fig. 5. Rule Assume just manifests the assumed formula ρ to the postcondition [26, Sec. VII B]. In comparison to the other three rules, there is no justification yet why this assumption can be made—as described in Section 2, this justification comes from the global declassification policy instead as formalized in Section 6, where we will also show the rule for command **trace** e.

The rule for commands **output** $\ell'\ e_v$, which outputs the value e_v to the attacker who is at security level ℓ', requires that the ℓ'-level attacker knows the value e_v being output and, hence, does not learn any new information. It also requires that the expression ℓ' denoting the level at which e_v is being output is known to the ℓ-level attacker since, otherwise, the choice of the level on which the output is occurring could leak information. In the example from Section 2, an output command is represented as library function `print_average()`, which specifies that its argument must be **`low`**.

In accordance with the threat model from Section 3, the rules for loading and storing via pointer e_p are similarly guarded to not leak information via the memory access pattern, by enforcing that the pointer is not sensitive.

4.3 Program Semantics

We briefly sketch how program execution is modeled by a typical small-step operational semantics, similarly to Vafeiadis' formulation [76], as we rely on this and extend it later. A *configuration* captures the runtime state of a program:

$$\text{configuration } k, k' ::= (\textbf{run}\ L, c, s, h) \mid (\textbf{stop}\ L, s, h) \mid (\textbf{abort})$$

The configuration (**run** L, c, s, h) represents a running program whose current state is given by the store s and heap h and whose remaining program to execute is the command c; L is the set of locks not currently acquired. The configurations (**stop** L, s, h) and (**abort**) represent respectively the (successfully) terminated program whose final unacquired locks are L and whose final state is the store s and heap h, and aborted programs (e.g. due to a memory violation). SecCSL associates to each lock a resource invariant, like the one shown in Eq. (1) in Section 2. We denote by $\mathsf{invs}(L)$ the conjunction of the invariants of locks in the set L, i.e., shared state that is not currently accessed in a critical section.

The transition relation $k \xrightarrow{\sigma} k'$ represents one execution step from configuration k to configuration k' producing the *schedule* σ. Schedules σ record the sequence of *actions* of the program that are relevant to capture the observational powers of the attacker under the threat model of Section 3. We denote by $k \xrightarrow{\sigma}{}^* k'$ the reflexive transitive closure of the transition relation. The action **L** (respectively **R**) represents the decision to schedule the left (respectively right) command in a parallel composition $\|$. The action τ represents the execution of atomic command like assignments. In the next sections we will extend this semantic model of actions to capture the program's input-/output behavior, those steps that correspond to assumptions made in the verification, and we will also make memory access explicit in the schedule.

Our security guarantees will be expressed relative to what an attacker can observe from the schedules of runs and whether information leaks are covered by policies. In Section 5 and Section 6 we will capture these notions formally. The complete set of semantic rules is in Appendix C, Fig. 10. In the next section we present those that are relevant to the our extensions of SecCSL.

While the noninterference guarantee of SecCSL [37, Theorem 2] focuses on comparing heap locations, we point out that $\vdash_\ell \{P\}\ c\ \{Q\}$ implies that for a given major execution (**run** L, c, s, h) $\xrightarrow{\sigma}{}^*$ k of program c ending in some final/intermediate configuration k, any minor run (**run** L, c, s', h) $\xrightarrow{\sigma'}{}^*$ k' of the same length, as encoded by $|\sigma| = |\sigma'|$, will satisfy that the attacker learns nothing from the schedule, which we relax to equivalence of observations below, written $\sigma \approx_\ell \sigma'$, after introducing additional types of actions into the schedule. Moreover, the matched rules enforce semantically that k and k' are either both final or both running configurations with the same program.

5 POLICY-AGNOSTIC GUARANTEE

In this section, we discuss the *policy-agnostic* part of our contribution. It is based on the notion of semantic *actions*, which give rise to *schedules* that are much more informative than those of vanilla SecCSL, which in turn allows us to reason about attacker knowledge gained from observing an execution of a program that exhibits such schedules (Definitions 5.1 and 5.3).

The grammar of actions is as follows

$$\text{action } a ::= \tau \mid \textbf{L} \mid \textbf{R} \mid \textbf{Out}\ \ell\ v \mid \textbf{Assm}\ s\ \rho \mid \textbf{Load}\ p \mid \textbf{Store}\ p \mid \textbf{Trace}\ e$$

$$\frac{}{\vdash_\ell \{P(e)\}\ x := e\ \{P(x)\}} \qquad \frac{}{\vdash_\ell \{P\}\ \textbf{lock}\ l\ \{P \star \mathsf{inv}(l)\}} \qquad \frac{}{\vdash_\ell \{P \star \mathsf{inv}(l)\}\ \textbf{unlock}\ l\ \{P\}} \qquad \frac{\vdash_\ell \{P_1\}\ c_1\ \{P_2\} \qquad \vdash_\ell \{P_1\}\ c_2\ \{P_2\}}{\vdash_\ell \{P_1\}\ c_1; c_2\ \{P_2\}}$$

$$\frac{\vdash_\ell \{P_1\}\ c_1\ \{Q_1\} \qquad \vdash_\ell \{P_1\}\ c_2\ \{Q_2\} \qquad \mathsf{fv}(c_i) \cap \mathrm{mod}(c_j) = \varnothing}{\vdash_\ell \{P_1 \star P_2\}\ c_1 \parallel c_2\ \{Q_1 \star Q_2\}} \qquad \frac{P \implies \phi :: \ell \qquad \vdash_\ell \{P \star \phi\}\ c_1\ \{Q\} \qquad \vdash_\ell \{P \star \neg\phi\}\ c_2\ \{Q\}}{\vdash_\ell \{P\}\ \textbf{if}\ \phi\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{Q\}}$$

$$\frac{P \implies \phi :: \ell \qquad \vdash_\ell \{P \star \phi\}\ c\ \{P\}}{\vdash_\ell \{P\}\ \textbf{while}\ \phi\ \textbf{do}\ c\ \{P \star \neg\phi\}} \qquad \frac{P \implies \phi :: \ell \qquad \vdash_\ell \{P\ \star \phi\}\ c\ \{Q\} \qquad \vdash_\ell \{P\ \star \neg\phi\}\ c\ \{Q\}}{\vdash_\ell \{P\}\ c\ \{Q\}}\ \textsc{Split}$$

Figure 4: Some rules of SecCSL, where $\mathsf{fv}(e)$ denotes free variables of e. Highlighted side-conditions represent security checks.

$$\frac{}{\vdash_\ell \{\textbf{emp}\}\ \textbf{assume}\ \rho\ \{\rho\}}\ \textsc{Assume}$$

$$\frac{}{\vdash_\ell \{\ e_v :: \ell' \star \ell' :: \ell\ \}\ \textbf{output}\ \ell'\ e_v\ \{\textbf{emp}\}}\ \textsc{Output}$$

$$\frac{x \notin \mathsf{fv}(\{e, e_p\})}{\vdash_\ell \{e_p\ \mapsto\ e\ \star e_p :: \ell\ \}\ x := [e_p]\ \{x = e \star e_p\ \mapsto\ e\}}\ \textsc{Load}$$

$$\frac{}{\vdash_\ell \{e_p\ \mapsto\ e\ \star e_p :: \ell\ \}\ [e_p] := e_v\ \{e_p\ \mapsto\ e_v\}}\ \textsc{Store}$$

Figure 5: Proof rules for commands that produce security-relevant actions; security checks are highlighted. Discussion of command trace e is deferred to Section 6.

$$(\textbf{run}\ L, \textbf{assume}\ \rho, s, h) \xrightarrow{\langle \textbf{Assm}\ s\ \rho \rangle} (\textbf{stop}\ L, s, h)$$

$$(\textbf{run}\ L, \textbf{output}\ \ell'\ e_v, s, h) \xrightarrow{\langle \textbf{Out}\ [\![\ell']\!]_s\ [\![e_v]\!]_s \rangle} (\textbf{stop}\ L, s, h)$$

$$(\textbf{run}\ x := [e_p], L, s, h) \xrightarrow{\langle \textbf{Load}\ [\![e_p]\!]_s \rangle} (\textbf{stop}\ L, s', h) \quad \text{for } s' = s\big(x := h([\![e_p]\!]_s)\big) \text{ if } [\![e_p]\!]_s \in \mathrm{dom}(h)$$

$$(\textbf{run}\ L, \textbf{trace}\ e, s, h) \xrightarrow{\langle \textbf{Trace}\ [\![e]\!]_s \rangle} (\textbf{stop}\ L, s, h)$$

Figure 6: Semantic rules for program execution.

where internal action τ and concurrent scheduling decision $\mathbf{L}$, $\mathbf{R}$ are inherited from SecCSL (cf. Section 4.3), and the new actions arise from the execution steps of the commands discussed in this section. The key transitions are shown in Fig. 6. Heap access through an invalid pointer instead produces an (**abort**) successor configuration (but still exposes the same action in the schedule). The rule for memory writes (stores) is analogous to that of loads. Formulated this way, the extra provisions or security in terms of the action labels do not in any way constrain the program execution, it just exposes the necessary information for a later analysis.

We define attacker knowledge based on the *observation* that they can make of a schedule and what can be learned to reduce one's *uncertainty* about possible initial states resp. secrets. The first main result, Theorem 5.6, formalizes the promise made in Section 2 that any gain in knowledge is linked to an earlier assumption failure.

Definition 5.1 (Attacker-visible actions and schedules). For an action a, $\mathrm{visible}_\ell(a)$ keeps a if it is visible to an ℓ attacker and erases it into τ otherwise. This definition is lifted to schedules in the obvious way as $\mathrm{visible}_\ell(\sigma)$.

$$\begin{aligned} \mathrm{visible}_\ell(\tau) &= \tau & \mathrm{visible}_\ell(\mathbf{L}) &= \mathbf{L} \\ \mathrm{visible}_\ell(\textbf{Assm}\ s\ \rho) &= \tau & \mathrm{visible}_\ell(\mathbf{R}) &= \mathbf{R} \\ \mathrm{visible}_\ell(\textbf{Trace}\ e) &= \tau & \mathrm{visible}_\ell(\textbf{Load}\ p) &= \textbf{Load}\ p \\ & & \mathrm{visible}_\ell(\textbf{Store}\ p) &= \textbf{Store}\ p \end{aligned}$$

$$\mathrm{visible}_\ell(\textbf{Out}\ \ell'\ v) = \textbf{Out}\ \ell'\ v \text{ if } \ell \sqsubseteq \ell' \text{ else } \tau$$

The concurrent schedule is always visible and so are memory accesses. An output action is visible only for an attacker who is allowed to observe the respective channel. In contrast, assumption steps are modeled to *not* be visible because they do not constitute actual observations, albeit an attacker with knowledge about the program's source code knows their occurrence and the assumed formula ρ, due to the fact that executions are always matched.

Definition 5.2 (Observably equivalent schedules). Two schedules σ and σ' are observably equivalent for an ℓ attacker, written $\sigma \approx_\ell \sigma'$, if their ℓ-visibility projection is the same:

$$\sigma \approx_\ell \sigma' \quad \hat{=} \quad \mathrm{visible}_\ell(\sigma) = \mathrm{visible}_\ell(\sigma')$$

Note that this implies that the length of σ and σ' is the same.

Information leakage is phrased in the standard *knowledge-based* style [5, 7, 16, 18]. This style of security property talks about the attacker's knowledge in order to state that the attacker doesn't learn anything that should not have been revealed to them. Knowledge is captured in terms of the attacker's *uncertainty* about the program's secret data that the attacker is not supposed to learn.

Specifically, uncertainty is the complement of knowledge so decreased attacker uncertainty corresponds to an increase in attacker knowledge. The following definition captures this intuition. Together with visible$_\ell$(_) it serves as the *formal specification* of the adversarial capabilities outlined in Section 3.

Definition 5.3 (Attacker Uncertainty). For a given initial state (s, h) and schedule σ for command c, the attacker must accept as explanations all possible initial states (s', h') which can produce an observably equivalent schedule σ':

$$\begin{aligned}&\mathsf{uncertainty}_\ell(P, \sigma, c, L, s, h) \quad \hat{=} \\ &\quad \{\ (s', h') \mid \exists\ \sigma'\ k'.\ (s, h), (s', h') \vDash P \star \mathsf{invs}(L) \\ &\qquad\qquad \wedge\ (\mathbf{run}\ L, c, s', h') \xrightarrow{\sigma'}{}^{*}\ k' \wedge \sigma \approx_\ell \sigma'\ \}\end{aligned}$$

The property assesses how the attacker's uncertainty changes over time. Specifically we can use it to compare the attacker's uncertainty before and after each execution step. Any decrease in uncertainty represents new information that the attacker learned from that step. The property requires that this change in knowledge must be bounded by what the attacker is permitted to learn by that step of execution: For the policy-agnostic guarantee, each execution step is allowed to reveal (i.e. decrease the attacker's uncertainty about) only failed **assume** ρ steps, which in turn correspond to unsatisfied **Assm** ρ actions in the schedule:

Definition 5.4 (Assumption failure). An assumption failure occurs at position n with $n < |\sigma|$ and $n < |\sigma'|$ in a pair of schedules, if at that point both contain the same assumption ρ, and that assumption is not satisfied between the associated stores recorded in the action.

$$\begin{aligned}&\mathsf{assumption\text{-}failed}_\ell(n, \sigma, \sigma') \\ &\quad \hat{=}\ \ \exists s\ s'\ \rho.\ \sigma_n = \mathbf{Assm}\ s\ \rho \text{ and } \sigma'_n = \mathbf{Assm}\ s'\ \rho \text{ and } s, s' \not\models_\ell \rho\end{aligned}$$

Complementary to uncertainty, we formalize what the ℓ-level attacker is allowed to learn from a single execution step following a known execution prefix from initial state (s, h) with schedule σ. This will be defined as a set assumed-release$_\ell(P, \sigma, c, L, s, h,)$ of initial states (s', h') which the attacker is allowed to *exclude* from their uncertainty by observing such an additional step.

Definition 5.5 (Release by assumption).

$$\begin{aligned}&\mathsf{assumed\text{-}release}_\ell(P, \sigma, c, L, s, h) \quad \hat{=} \\ &\quad \{\ (s', h') \mid \exists\ \sigma'\ k'.\ (s, h), (s', h') \vDash P \star \mathsf{invs}(L) \\ &\qquad\qquad \wedge\ (\mathbf{run}\ L, c, s', h') \xrightarrow{\sigma'}{}^{*}\ k' \wedge \sigma \approx_\ell \sigma' \\ &\qquad\qquad \wedge\ \exists\ n.\ n < |\sigma| \wedge n < |\sigma'| \\ &\qquad\qquad\qquad \wedge\ \mathsf{assumption\text{-}failed}(n, \sigma, \sigma')\ \}\end{aligned}$$

This definition mirrors Definition 5.3 except that only those initial states (s', h') are kept that can lead to a failed assumption.

Theorem 5.6 (Policy-agnostic security guarantee). *If* $\vdash_\ell \{P\}\ c\ \{Q\}$ *then for a major run* $(\mathbf{run}\ L, c, s, h) \xrightarrow{\sigma_1}{}^{*} k_1$ *the knowledge gain from one additional step* $k_1 \xrightarrow{\sigma_2} k_2$*, expressed as the difference in uncertainty, is bounded by the release condition:*

$$\begin{aligned}&\mathsf{uncertainty}_\ell(P, \sigma_1, c, L, s, h) \setminus \mathsf{uncertainty}_\ell(P, \sigma_1 \cdot \sigma_2, c, L, s, h) \\ &\subseteq \mathsf{assumed\text{-}release}_\ell(P, \sigma_1, c, L, s, h)\end{aligned}$$

6 CONFORMANCE WITH POLICIES

With assumed-release$_\ell$(_) we have a precise characterization of possible information leaks due to failed assumptions. Next we show how these leaks can be justified and formally bounded in terms of high-level declassification policies:

Definition 6.1 (Declassification policy). A declassification policy $\mathcal{D}(tr) = \varphi_\mathcal{D}(tr) \rightsquigarrow \rho_\mathcal{D}(tr)$ specifies a condition $\varphi_\mathcal{D}$ that states *when* the policy applies and a relational release formula $\rho_\mathcal{D}$ that encodes *what* information is allowed to be released then.

Both constituents $\varphi_\mathcal{D}$ and $\rho_\mathcal{D}$ may mention the current trace tr: *List*⟨*Event*⟩, a logical list of application-specific *Event*s. In the example from Section 2, events are just the numbers returned from and added to the trace by `avg_get_input()`.

It is sometimes convenient (cf. Section 7) to let the formulas range over common auxiliary parameters $\vec{x}$, where $\varphi_\mathcal{D}(tr, \vec{x}) \rightsquigarrow \rho_\mathcal{D}(tr, \vec{x})$, abbreviates the slightly involved policy $(\exists\, \vec{x}.\ \varphi_\mathcal{D}(tr, \vec{x})) \rightsquigarrow (\forall\ \vec{x}.\ \varphi_\mathcal{D}(tr, \vec{x}) \implies \rho_\mathcal{D}(tr, \vec{x}))$. The intuitive reading is just that the condition may bind some values that are later referred to by the release which encodes a policy that declassifies different information depending on a number of cases in $\varphi_\mathcal{D}$.

In order to track the trace tr throughout the verification, we extend the assertions by a designated abstract history predicate $\mathcal{H}$(_), and for the sake of presentation we also introduce an explicit mechanism to extend this trace by an additional command **trace** e (in Verdeca this is instead realized as library annotations), where expressions tr and e denote a trace resp. event,

$$\text{assertion } P \ ::= \ \cdots \mid \mathcal{H}(tr) \qquad \text{command } c \ ::= \ \cdots \mid \mathbf{trace}\ e$$

where $\mathcal{H}(tr)$ says that current value of expression tr is the trace until now, and **trace** e is a specification command that extends this trace by an additional event, denoted by expression e.

The purpose of an audit of a given verification with respect to a policy is to inspect each **assume** ρ statement placed in the program. To that end, we need to refer to the *verification context* at that point, specifically the assertion/path condition P and trace tr that occurs in the sub-derivation $\vdash_\ell \{P \star \mathcal{H}(tr)\}$ **assume** $\rho; \ldots\ \{\ldots\}$ of that program part. A policy $\mathcal{D}$ is honored if P implies $\varphi_\mathcal{D}$ and $\rho_\mathcal{D}$ with P implies ρ at every such occurrence of assumptions.

In the same spirit as the rest of the paper, we show how the concern of policy adherence can be separated out of the verification of the program implementation (cf. comments at the end of Section 2) in such a way that we can still draw a connection between all respective constituents. We define *extended* judgements

$$\vdash_\ell \{P \star \mathcal{H}(tr)\}\ c\ \{Q \star \mathcal{H}(tr')\} \triangleright A$$

that now mention explicitly the history predicate which is threaded through the proof alongside all other assertions [14, 36, 64, 72]. Moreover, we instrument the proof rules to produce a set A of *audit triples* (P, tr, ρ) from each occurrence of an assumption as the verification context mentioned above.

Some interesting proof rules are shown in Fig. 7: Emitting a trace event e symbolically extends the trace expression bound by history predicate $\mathcal{H}$(_) to $tr \cdot \langle e \rangle$. Assumptions produce an audit triple that records the current proof context P, tr alongside the assumed formula ρ. As an example for syntax-directed rules, sequential composition merges the results from both commands. Rule Frame

$$\frac{}{\vdash_\ell \{\mathcal{H}(tr)\}\ \textbf{trace}\ e\ \{\mathcal{H}(tr \cdot \langle e \rangle)\} \triangleright \varnothing}\textsc{Emit}$$

$$\frac{}{\vdash_\ell \{P \star \mathcal{H}(tr)\}\ \textbf{assume}\ \rho\ \{P \star \rho \star \mathcal{H}(tr)\} \triangleright \{(P, tr, \rho)\}}\textsc{Assume}$$

$$\frac{\vdash_\ell \{P\}\ c_1\ \{Q\} \triangleright A_1 \qquad \vdash_\ell \{Q\}\ c_2\ \{R\} \triangleright A_2}{\vdash_\ell \{P\}\ c_1; c_2\ \{R\} \triangleright A_1 \cup A_2}\textsc{Seq}$$

$$\frac{\vdash_\ell \{P\}\ c\ \{Q\} \triangleright A \qquad \mathsf{mod}(c) \cap \mathsf{fv}(F) = \varnothing}{\vdash_\ell \{P \star F\}\ c\ \{Q \star F\} \triangleright \{(P \star F, tr, \rho) \mid (P, tr, \rho) \in A\}}\textsc{Frame}$$

Figure 7: Proof rules for event histories and audit triples.

shows that any frame condition F that is preserved by the execution of c can be adjoined to the audit triples after the fact, such that alternatively rule Assume could have been formulated as a "small axiom" with **emp** instead of a general P, i.e., framing is compatible with recording proof contexts. Now we can formalize policy audit.

Definition 6.2 (Policy audit). A verification $\vdash_\ell \{P \star \mathcal{H}(tr)\}\ c\ \{Q \star \mathcal{H}(tr')\} \triangleright A$ is correctly audited wrt. a policy $\mathcal{D}(tr) = \varphi_{\mathcal{D}}(tr) \rightsquigarrow \rho_{\mathcal{D}}(tr)$ if for each $(P, tr, \rho) \in A$ implications $P \implies \varphi_{\mathcal{D}}(tr)$ and $P \star \rho_{\mathcal{D}}(tr) \implies \rho$ are valid.

Intuitively, audit triples are simply proof obligations for every assumption to be justified by the declassification policy. As with the policy-agnostic security guarantee, we now provide a semantic guarantee that bridges between A from the calculus and information release by policy. We denote by $\mathsf{trace}(\sigma)$ the sequence of values e from **Trace** e actions in the schedule σ, defined as $\mathsf{trace}(\sigma) \triangleq \langle e \mid \textbf{Trace}\ e \in \sigma \rangle$ with the intention that $\mathsf{trace}(\sigma)$ coincides with the evaluation of the trace expression tr in any post-state that asserts $\mathcal{H}(tr)$. Moreover, as $\varphi_{\mathcal{D}}(\_)$ can be regarded a formula of one variable, say $\mathtt{tr}$, we write $\sigma \models \varphi_{\mathcal{D}}$ when $[\![\varphi_{\mathcal{D}}(\mathtt{tr})]\!]_s$ is true for state $s(\mathtt{tr}) = \mathsf{trace}(\sigma)$ (all other variables in s are irrelevant), similarly, we write $\sigma, \sigma' \models \rho_{\mathcal{D}}(\mathtt{tr})$ for $s, s' \models \rho_{\mathcal{D}}(\mathtt{tr})$ and $s(\mathtt{tr}) = \mathsf{trace}(\sigma)$, $s'(\mathtt{tr}) = \mathsf{trace}(\sigma')$. We define counterparts to assumption-failed and assumed-release with respect to policies.

Definition 6.3 (Policy exclusion). A declassification policy $\mathcal{D} = \varphi_{\mathcal{D}} \rightsquigarrow \rho_{\mathcal{D}}$ excludes a pair of schedules (from the obligation to prove absence of leaks), if after some number of steps n with $n < |\sigma|$ and $n < |\sigma'|$ the declassification condition is satisfied but the release formula is not:

$$\begin{aligned}&\mathsf{policy\text{-}excludes}_\ell(\varphi_{\mathcal{D}} \rightsquigarrow \rho_{\mathcal{D}}, n, \sigma, \sigma') \triangleq \\ &\quad \sigma_{|n} \models \varphi_{\mathcal{D}} \text{ and } \sigma'_{|n} \models \varphi_{\mathcal{D}} \text{ and } \sigma_{|n}, \sigma'_{|n} \not\models \rho_{\mathcal{D}}\end{aligned}$$

Definition 6.4 (Release by policy). The initial states (s', h') that an attacker may remove from their uncertainty by observing any further step after a prefix run with schedule σ are those minor runs that are excluded by policy $\mathcal{D}$.

$$\begin{aligned}&\mathsf{policy\text{-}release}_\ell(\mathcal{D}, P, \sigma, c, L, s, h) \triangleq \\ &\quad \{\ (s', h') \mid \exists\ \sigma'\ k'.\ (\textbf{run}\ L, c, s', h') \xrightarrow{\sigma'}{}^{*}\ k' \wedge \sigma \approx_\ell \sigma' \\ &\qquad \wedge\ \exists\ n.\ n < |\sigma| \wedge n < |\sigma'| \wedge \mathsf{policy\text{-}excludes}(\mathcal{D}, n, \sigma, \sigma')\ \}\end{aligned}$$

Finally, we can state the second main result:

Theorem 6.5 (Policy-specific security guarantee). *For a verified program $\vdash_\ell \{P \star \mathcal{H}(\langle\rangle)\}\ c\ \{Q \star \mathcal{H}(tr')\} \triangleright A$ and a policy D formally audited according to Definition 6.2, for each major run $(\textbf{run}\ L, c, s, h) \xrightarrow{\sigma_1}{}^{*}\ k_1$ with final step $k_1 \xrightarrow{\sigma_2} k_2$ we have:*

$$\begin{aligned}&\mathsf{assumed\text{-}release}_\ell(P, \sigma_1, c, L, s, h) \\ &\subseteq \mathsf{policy\text{-}release}_\ell(\mathcal{D}, P, \sigma_1, c, L, s, h)\end{aligned}$$

Under the conditions of Theorem 6.5, we get (owing to Theorem 5.6) the ultimate property that every knowledge increase is within the policy:

$$\begin{aligned}&\mathsf{uncertainty}_\ell(P, \sigma_1, c, L, s, h) \setminus \mathsf{uncertainty}_\ell(P, \sigma_1 \cdot \sigma_2, c, L, s, h) \\ &\subseteq \mathsf{policy\text{-}release}_\ell(\mathcal{D}, P, \sigma_1, c, L, s, h)\end{aligned}$$

7 CASE STUDIES

We demonstrate the approach of this paper with several challenging case studies that we implemented and verified using auto-active verifier Verdeca, an extension of SecC that adds constant-time security checks for memory access, adapts the semantics of value classification as described in Section 4, and adds an `-audit` flag which shows the audit conditions from Section 6 to the user. (In all case studies we inline the checks of these audit obligations into the verification, as explained at the end of Appendix A, so that Verdeca discharges them automatically.) Verdeca is implemented in the Scala programming language and encompasses roughly 5 KLoC. Like SecC, Verdeca mechanises the application of our extended SecCSL logic (i.e., the application of the rules in Fig. 4 and Fig. 5) by symbolic execution. The case-studies are written in C with logical definitions and program annotations formulated in the specification language of SecC. Verdeca inherits some limitations from SecC: It supports a significant fragment of C but lacks for example union types, taking pointers to local variables, and floating point and bit-wise operations. Numeric types are treated as unbounded mathematical integers (as is common in auto-active verifiers) so that the verification is not sound in the presence of overflows. The absence of overflow can be proved in Verdeca by adding assertions on integer operations.

The formal soundness theorems Theorem 6.5 and Theorem 5.6, mechanised in Isabelle/HOL, apply to the simple command language formalised in this paper (whose semantics is given in Fig. 10). So Verdeca's soundness follows from those theorems, so long as Verdeca correctly implements the semantics of its subset of C and correctly implements the rules of Fig. 4 and Fig. 5. Aside from treating `int`s as unbounded integers we believe Verdeca is faithful to the formal program semantics and correctly implements the logic. Both of these assumptions could in principle be discharged by applying orthogonal ideas on validating the output of auto-active verifiers [45, 63].

Case Study	Proof Ratio	Verified SLOC	Unverified SLOC	Effort (pw)
Location Service	1.9	210	124	3
Auction Server	4.3	187	79	3.5
Wordle	5.9	47	81	0.3
Private Learning	2.5	315	114	6

Table 1: Case study statistics. We report the size of the case studies in Source Lines of Code (SLOC), including the size of the Verified code; the Unverified code; and the Proof Ratio, the ratio of the size of the proof (definitions, lemmas, specifications etc. as VERDECA annotations) to that of the verified code. We also report the total Effort in person-weeks (pw).

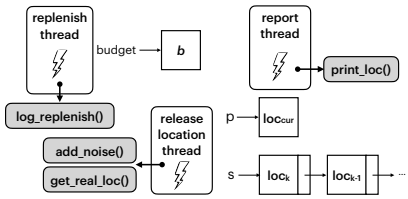

Figure 8: Architecture of the location service. External functions are coloured grey.

7.1 Differentially-Private Location Service

Our first case study implements a multi-threaded, privacy-preserving *location service*. Such a service might run, for instance, on a user's mobile phone. Its intention is to release information about the user's location, but in accordance with a privacy policy that implements differential privacy [33] for mobility traces [21] (i.e. traces of reported locations for the user).

This service, whose architecture is depicted in Fig. 8, contains three threads, that all run continuously: the *release location* thread periodically releases information about the user's location, in accordance with a differential privacy policy. It copies information about the user's most recent physical location, obtained via the external function `get_real_loc()`, into the heap location pointed to by `p`, after adding noise to ensure differential privacy, using the external function `add_noise()`. This function is an off-the-shelf implementation of the *planar laplacian* [4] geolocation privacy mechanism. The state of the differential privacy policy is recorded in the location pointed to by `budget`. Finally, this thread maintains a (linked) list `s` carrying information about the user's prior locations.

This linked list `s` is used in situations where the privacy budget has been exhausted. In particular, each time noise is added to the user's real, private location via the external `add_noise()` function, some of the user's privacy budget is consumed. When the budget gets sufficiently low, no new information about the user's location can be revealed without violating differential privacy. In this situation, the *release location* thread instead applies extrapolation to *predict* [21] the user's most likely current location from the location history recorded in the linked list `s`. Importantly, this history contains only values already previously released, i.e. noisy values resulting from previous applications of the differential privacy mechanism `add_noise()`. Hence, releasing a location prediction made from those values reveals no new information, while still allowing the *release location* thread to provide continuous service.

The *report thread* periodically reports the user's most recent location as recorded by the *release location* thread in `p`. Finally, the *replenish budget* thread periodically replenishes the privacy budget `budget`, allowing the *release location* thread to again apply the differential privacy mechanism to release new (noisy) location points (rather than predictions). A global lock controls access to all shared data structures: `p`, `s` and `budget`.

Security Policy. The policy ensures that the differential privacy mechanism is appropriately applied and that no raw location data (to which the mechanism has not been applied) is ever revealed. Specifically, the external function `print_loc()` has a contract that requires its argument is **low**; yet the contract for `get_real_loc()` says that the returned location is **high** so that in the absence of declassification, no location data can ever be revealed by the service.

The declassification policy allows releasing location data only via the correct use of the differential privacy mechanism. Traces *tr* in this example are sequences of events, each of which is either an event **Consumed** (*nlat*, *nlon*), recording that the differential privacy mechanism produced a noisy location point (*nlat*, *nlon*) and consuming a fixed positive amount ϵ of the privacy budget; or the event **Replenished** recording that the privacy budget was replenished by a fixed positive amount r. Both ϵ and r are public (**low**) constants that control the strength of the privacy guarantee.

The contract for the external function `add_noise()` that implements the differential privacy mechanism is as follows. For brevity we elide, via ... in the precondition, that the argument `pt` is a valid pointer to a `struct point`, and freely intermix ASCII and mathematical notation.

```
void add_noise(struct point *pt);
_(requires ∃tr. H(tr) ...)
_(ensures  ∃nlat nlon. H(tr · Consumed (nlat, nlon)))
_(ensures  &pt->lat ↦ nlat ⋆ &pt->lon ↦ nlon)
```

Whenever this function is called an appropriate event is recorded in the trace to remember that some privacy budget was consumed. Similarly, **Replenished** events are generated by the external function `log_replenish()`, which is called by the *replenish* thread when replenishing the budget.

The declassification policy $\mathcal{D}_{\text{loc}}(tr, e) = \varphi_{\text{loc}}(tr, e) \rightsquigarrow \rho_{\text{loc}}(tr, e)$ then specifies that it is safe to declassify the value e only when e is the most recent point (*nlat*, *nlon*) generated by the differential privacy mechanism `add_noise()` only when there is sufficient privacy budget.

$$\varphi_{\text{loc}}(tr, (nlat, nlon)) \triangleq \mathcal{H}(tr) \star \text{count\_budget}(tr) \geq \epsilon \star \exists\, tr'.\ tr = tr' \cdot \textbf{Consumed}\ (nlat, nlon)$$

$$\rho_{\text{loc}}(tr, (nlat, nlon)) \triangleq (nlat, nlon) :: \textbf{low}$$

The function count_budget() (defined by straightforward recursion on *tr*) iterates through *tr* to count up the budget remaining at the present time—consume events add $-\epsilon$ while replenish events add r to the budget, which starts at 0.

This security policy demonstrates how our approach securely handles multiple (or repeated) declassifications in the same execution of a single program: each act of declassification is justified

separately by appealing to the policy wrt. the trace *tr* at that point in time.

It is important to note that this security policy does not verify that the differential privacy mechanism has been appropriately *implemented*; instead it verifies that it is appropriately *used* by the application. In other words, this policy does not directly state a differential privacy guarantee; however it ensures such a guarantee under the assumption that the differential privacy mechanism correctly provides the privacy guarantee represented by the deplenishing budget. Verifying the mechanism implementation of course requires probabilistic reasoning and is best done using other approaches [10]. Similar arguments apply to the private learning case study later on (Section 7.4).

Verification. Since Verdeca does not currently support reasoning about floating point arithmetic, for this case study such arithmetic (e.g. in the verified extrapolation code) is modelled using integer arithmetic instead. This is sound since the program invariant and security policy do not depend on any floating-point arithmetic. We prove the verified extrapolation code operates only on public values, ruling out timing channels from floating point operations [47]. The verified part of this application comprises 210 source lines of code (SLOC). Unverified code (124 SLOC) comprises the external functions that allocate memory, acquire and release the global lock, implement the planar laplacian mechanism (72 SLOC), and simulate generating user location points, as well as the `main()` function that sets up the threads. The proof:code ratio (proof SLOC to verified code SLOC) is 1.9. This effort to complete this case study was approximately 3 person-weeks (see Table 1).

7.2 Sealed-Bid Auction Server

Our second case study is a *sealed-bid* auction server. In such an auction, all bids are kept secret until the auction is complete. This prevents bidders racing to outbid one another. Our server uses concurrency to ensure that no bidder can deny service to another by servicing each client connection in a separate thread.

Each client bid is handled by a separate *handle bid* thread. A separate *close auction* thread waits until the auction duration (a fixed, public parameter) has elapsed and then closes the auction. Both make use of external logging functions: the *handle bid* thread logs incoming bids using the `log_bid()` external function, while the *close auction* thread logs the fact that the auction has closed using the `log_closed()` function. Logging is important in this case study to provide an audit trail (e.g. in case of a disputed auction). The *close auction* thread uses the external `print_result()` function to print out the result of the auction, once it has been closed.

When receiving a new bid, the *handle bid* thread compares the newly submitted bid to the current maximum in a constant-time fashion, and updates the latter if the new bid is larger. Bids are pairs (id, qt) where *id* is the identity of the client who submitted the bid and *qt* is the amount (or *quote*) of the bid.

Security Policy. The top-level verified function that implements the *handle bid* thread takes as its argument the bid to be handled. The precondition on this function states that the bid is secret (**high**). Thus all bids are treated as secret. The precondition on the external `print_result()` function requires that its argument is public (**low**). So the only way for a winner to be announced is via declassification. Verdeca's constant-time guarantee meanwhile ensures that no information about bids can be leaked (including via timing channels) prior to declassification.

The declassification policy states that no bid information can be declassified until after the auction is closed. At this time the only bid that can be declassified is the maximum bid that was received (i.e. the auction winner). Hence, the policy allows only the winning bid to be revealed only after the auction has closed, when the attacker learns the winning bid and that no other bid was higher.

To capture this policy, the trace *tr* records two kinds of events: **Run** (id, qt) represents the submission of a bid from the client with id *id* and amount *qt* while the auction is running; **Fin** represents that the auction has been closed. Each is generated by one of the external logging functions: **Run** (id, qt) is generated by the `log_bid()` function, while **Fin** is generated by `log_closed()`. In this way we piggy back on the application's normal functioning to define its security policy. The contracts for these external functions are similar to those for `add_noise()` and `log_replenish()` from Section 7.1.

The declarative, extensional declassification policy $\mathcal{D}_{\mathsf{bid}}(tr, e) \triangleq \varphi_{\mathsf{bid}}(tr, e) \rightsquigarrow \rho_{\mathsf{bid}}(tr, e)$ is defined as follows, where *e* is the value (id, qt) to be declassified.

$$\varphi_{\mathsf{bid}}(tr, (id, qt)) \triangleq \mathcal{H}(tr) \star \exists tr'.\ tr = tr' \cdot \mathbf{Fin} \star \mathsf{contains}(tr', \mathbf{Run}\ (id, qt)) \star \mathsf{ismax}(tr', qt)$$
$$\rho_{\mathsf{bid}}(tr, (id, qt)) \triangleq (id, qt) :: \mathbf{low}$$

where $\mathsf{contains}(xs, y)$ is the standard list function for testing whether list *xs* contains the element *y*, and $\mathsf{ismax}(tr', qt)$ checks that tr' contains only events **Run** (id', qt') for which $qt \geq qt'$ and is defined via recursion on tr'.

Policy Composition. To evaluate our approach's ability to handle the *composition* of multiple security policies, we decided to extend the example and augment its security policy. Specifically, we added a *reserve price* feature to the auction server. When run in this mode, the user supplies a (secret) reserve price and in order for a winner to be declared, there must be a bid that is greater-or-equal to this reserve. We parameterise our verification by an arbitrary, constant reserve price *r* and use the abstract separation logic predicate $Reserve(r)$ to denote that the auction is running in the reserve price mode and that *r* is the reserve price.

In this mode, all bidders learn whether any bid was $\geq$ the reserve since, if it was not, no winner is announced. Therefore the policy allows this (boolean) fact *met* to be declassified unconditionally (only) once the auction is closed.

$$\varphi_{\mathsf{met}}(tr, met) \triangleq \mathcal{H}(tr) \star \exists\ tr', r.\ Reserve(r) \star tr = tr' \cdot \mathbf{Fin} \star met = \mathsf{resmet}(tr', r)$$
$$\rho_{\mathsf{met}}(tr, met) \triangleq met :: \mathbf{low}$$

$\mathsf{resmet}(tr', r)$ is a simple recursive function that iterates through tr' returning **true** as soon as it finds a bid (id, qt) for which $qt \geq r$, or **false** if none is found. The declassification policy for the winning bid is then specified as follows, using φ_{bid} and ρ_{bid} defined for the

non-reserve mode earlier.

$$\varphi_{\mathsf{resbid}}(tr, (id, qt)) \mathrel{\hat{=}} \mathcal{H}(tr) \star \mathit{Reserve}(r) \star qt \geq r \star \varphi_{\mathsf{bid}}(tr, (id, qt))$$

$$\rho_{\mathsf{resbid}}(tr, (id, qt)) \mathrel{\hat{=}} \rho_{\mathsf{bid}}(tr, (id, qt))$$

The composed declassification policy is of course the non-overlapping disjunction of the mutually-exclusive predicates $\mathcal{D}_{\mathsf{met}}$ and $\mathcal{D}_{\mathsf{resbid}}$, depending on the type of the second argument.

Verification. The total size of the verified code for this application is 187 SLOC. Unverified code (79 SLOC) comprises external functions that allocate memory, acquire and release the global lock, implement logging and printing, as well as the code that reads from client socket connections, and the `main()` function that sets up the threads and the TCP listen socket. The size of the verified artifact for this case study is 985 source lines, making the proof:code ratio 4.3. This ratio is higher than previous because this case study involves a lot of meta-level reasoning by induction about the policy itself. This case-study required approximately 3 person-weeks of effort; adding the reserve price feature added 4 person-days (see Table 1).

7.3 Wordle

Our third case study is a constant-time implementation of the popular game Wordle. The implementation is a simple server that allows players to connect and to guess a pre-chosen 5-letter word w. In response to a player submitting her guess g, the server replies with a 5-byte response r. The ith byte r_i of the response provides information about the ith letter g_i of the guess in relation to the pre-chosen word w: 0 (black) indicates that g_i is not present anywhere in w; 1 (yellow) that it is present in w at some index j for which $g_j \neq w_j$; 2 (green) that $g_i = w_i$. The server runs a separate thread to service each client connection.

Security Policy. The security policy says that the pre-chosen word w is secret: $w :: \mathbf{high}$. Each player p is assigned a distinct security level ℓ_p. A player's guess g is known only to herself: $g :: \ell_p$, encoded in the postcondition of the external library function that retrieves the player's guess. The external function that transmits the server's response back to the player requires in its precondition that the response r is allowed to be known to the player: $r :: \ell_p$. Since r is a function of w, this requires declassification.

The policy condition $\varphi_{\mathsf{word}}(tr)$ for the declassification policy therefore requires that the player has submitted a most recent guess g, for the pre-chosen word w, which is encoded in the trace by appropriate events that record together for each submitted guess g the player p who submitted it as well as the guess g itself. An abstract separation logic predicate is used to remember which is the pre-chosen word w (similar to the *Reserve* predicate of Section 7.2)

The policy release formula $\rho_{\mathsf{word}}(tr)$ is more interesting. It specifies what information player p (with security level ℓ_p) is allowed to learn after submitting guess g for the pre-chosen word w, and requires that this information is not revealed to anyone else. Its core is the following:

$$\forall i.i :: \mathbf{low} \implies (i \geq 0 \wedge i < \mathsf{length}(w) \implies (w_i = g_i) :: \ell_p) \star \mathsf{ccontains}(w, g, g_i, \mathsf{length}(w)) :: (w_i \neq g_i ? \ell_p : \mathbf{high})$$

The first conjunct says the player is allowed to learn whether each letter g_i of the guess is equal to the corresponding letter of the word w_i. The second conjunct says that additionally, if $g_i \neq w_i$, then the player is allowed to learn whether g_i is contained elsewhere in the word at some location j for which $w_j \neq g_j$. That is the result returned by the function $\mathsf{ccontains}(w, g, g_i, \mathsf{length}(w))$ which is defined by straightforward recursion on the length of the word. Specifically, $\mathsf{ccontains}(w, g, c, n)$ returns true if c is contained in the first n characters of w at some location j where $w_j \neq c$.

Note that this security guarantee ensures the server does not leak information in its timing behaviour about the player's guess, which might otherwise be exploited by other players to draw extra inferences about the word w beyond what they could deduce from their guesses alone.

Table 1 reports the size and effort for this case study. The significantly higher proof:verified code ratio is because these proofs contain a large amount of generic meta-level reasoning (e.g. about lists and strings, etc.) required for this case study.

7.4 Private Learning

Our fourth case study investigates the application of our ideas to secure, private learning. We consider a scenario in which a client wishes to compute a model, possibly in collaboration with others, over very sensitive data (e.g. parental income, race, and gender to predict earnings distributions, incarceration rates [22–24], survival prediction of lung cancer patients [31], etc.). It is common for such models to be computed in hardware-supported *secure enclaves* [42, 43, 48, 55, 75] provided by trusted execution environments like Intel SGX [54] and ARM TrustZone [3], to defend against data theft including against the host operating system. Here, Verdeca's constant-time guarantee is especially relevant, given that TEEs are known to leak data via various side-channels [15, 49, 56, 57, 67]; enforcing constant-time ensures side-channels cannot be exploited.

In this case study, a client is invoked with initial model parameters θ^0. It runs T training iterations. At each iteration t ($1 \leq t \leq \mathsf{T}$) it refines the model parameters, producing new ones θ^{t+1}. It does so by applying differentially-private gradient descent [1] (DP-GD), in which a noisy gradient against the model's loss function is computed and then used to refine the model parameters. The goal is to ensure that the refined model $\theta^{\mathsf{T}+1}$ does not leak too much information about the sensitive training data. For simplicity, our current implementation learns a linear model over the training data. We refer to the process in which $\theta^{\mathsf{T}+1}$ is computed from θ^0 as a training *epoch*, comprising T training iterations.

Each training iteration consumes ϵ privacy budget; by composition, each epoch consumes $\mathsf{T} \cdot \epsilon$. As in the location service case study (Section 7.1), a separate thread may periodically replenish that budget, if desired. This design allows the client to be deployed in a federated learning setup in which clients periodically provide their updated model parameters $\theta^{\mathsf{T}+1}$ to a central server, which then e.g. computes the average across all client models, before sending that average back to each client to use as θ^0 for a subsequent training epoch. Distrusting clients can thus compute a shared model without revealing their sensitive data to each other, nor the server.

Security Policy. The initial model parameters θ^0 are $\mathbf{low}$, but the client's training data over which the updated parameters $\theta^{\mathsf{T}+1}$ are computed are **high**. The updated parameters are required to be $\mathbf{low}$, and so must be declassified.

Similarly to the location service (Section 7.1), traces record events to remember when ϵ privacy budget is consumed (on each training iteration) and when the budget is replenished. They also record events to remember when initial model parameters are received by the client at the start of each epoch, and when updated model parameters are released by the client at the end. Hence the declarative policy says that the updated model parameters can be declassified only when they have been correctly computed (T training iterations have occurred in the most recent epoch), for which there was sufficient privacy budget $\mathsf{T} \cdot \epsilon$ available before the epoch began.

Verification. The verified part of this case study comprises 315 SLOC, and 114 unverified SLOC whose functionality is similar to the prior case studies. The total verified artifact comprises 1108 source lines, yielding a proofs:code ratio of 2.5. No effort was made to optimise this ratio and indeed these proofs contain a certain amount of duplicated lemmas from other case studies.

8 CONCLUSION AND RELATED WORK

We presented a principled methodology for proving secure declassification for non-trivial, concurrent, programs. We decompose the problem into (a) proving that the program only leaks information it has explicitly declassified (via **assume** statements); and (b) auditing the declassifications against a declarative security policy $\mathcal{D}$ to ensure that all leaks accord with the policy. We provide a sound program logic, supported by the auto-active verifier Verdeca and applied it to reason about the implementations of various case studies on the order of hundreds of source lines of code.

In practice, one can of course choose to inline the policy audit (Definition 6.2) into the verification (this is illustrated at the end of Appendix A), or alternatively represent the declassification step that appeals to a policy by a specification-only procedure with precondition $\varphi_{\mathcal{D}}$ and postcondition $\rho_{\mathcal{D}}$; or alternatively to place the respective audit conditions into the code. By disentangling contributions (a) and (b) in our formal development we contribute a justification for this kind of reasoning with respect to a semantic characterization of attacker knowledge [26]. Similarly, our ideas are not necessarily tied to the presentation as an extension of the specific foundation SecCSL. With the appropriate care to semantic variations (e.g. timing-sensitivity), we think it is feasible to adapt the approach to other foundations like modular product programs [35] as implemented in Viper.

Prior work on practical secure declassification includes the verification of the kernel of a conference management system [66], a social media platform [12] and its distributed successor [11]. These works proved variants of the generic security property of Bounded Deducibility [65], which is similar to declassification policies $\mathcal{D}$. The proofs use manual unwinding in Isabelle/HOL, over an abstract program representation of I/O automata. Li et al. [52] verified secure declassification policies while verifying a 3.8K SLOC Linux KVM hypervisor, in the proof assistant Coq. Their policies were encoded non-declaratively by artificially modifying the semantic model to replace declassified sensitive data with non-sensitive data, allowing declassification to be proved in terms of standard noninterference.

Banerjee et al. [9] enhance the knowledge-based security property of Askarov and Sabelfeld [7] with relational assumptions (not using that term), and propose enforcement using a security type system together with relational verification of the declassifying code. Their declassification policies combine the assumption with an assertion, which should refer to ghost state modeling external observations. Their formalization is for deterministic sequential programs and does not include the requisite relational logic. We show the approach can be applied to concurrent programs as well. We decouple meaning of assume and meaning of policies (cf. their Def 5.5), such that assume statements have meaning independently of a stated policy. Our proof system (Section 4) and audits (Section 6) provide a way to formally establish the requirements outlined by their Definition 6.2 points 2 and 3. Our explicit trace predicate $\mathcal{H}(tr)$ realizes their suggested ghost state. By contrast with their suggestion to use a type system for some relational reasoning, we use only the proof system, which encompasses relational reasoning.

Balliu et al. [8] observe that knowledge-based properties like these are closely related to standard semantics of epistemic logic, and show how several properties from the literature can be expressed in epistemic temporal logic (but this work does not address verification of such properties, nor concurrent programs).

Askarov et al. [6] formulate knowledge-based security for monitoring of concurrent programs with synchronization in the form of barriers; their monitor is hybrid in the sense that it relies on an oracle for static analysis of branches not taken. Compared with a logic or static analysis, monitoring has the advantage that it can allow use of a program under conditions when its execution is secure, even if the program is not secure in general. Monitoring has the disadvantage of significant runtime overhead. Owing to nuanced use of rely-guarantee reasoning and annotations that designates assumptions a thread makes about locality of shared variables (adapted from Mantel et al. [53]), their monitor is factored into local and global parts and avoids the need for additional synchronization.

There is an extensive literature on verification of constant-time security properties; a recent example is Shivakumar et al. [73] which also addresses the role of compilers in mitigation. Language based mitigations of timing channels have been studied since Russo [69].

There is also an extensive literature on information flow for concurrent programs. Prior to the emergence of knowledge-based formulations many variations were based on specialized bisimulations (e.g., Sabelfeld and Sands [70]). There are tradeoffs between permissiveness and compositionality of the different properties (see e.g., Mantel et al. [53]), and differing models are of interest depending on adversary models. Surprisingly the property of Sabelfeld and Sands [70] is decidable [30] provided the data model is sufficiently simple for the expression language to be decidable.

The Veronica logic proves secure declassification for shared-memory concurrent programs [72]. Its security property is also formulated as a knowledge-based one. It is more permissive than our constant-time property in that it can tolerate some secret-dependent branches. However, to avoid occlusion anomalies [71], such branching is disallowed for secrets involved in declassification.

Veronica supports only *unary* (non-relational) assertions, the entire logic is designed around and fundamentally tied to this principle. Lacking relational assertions like $e :: \ell$, however, limits its expressiveness and precludes scalability. Instead of writing $e :: \mathbf{low}$ for example, in Veronica one has to precisely specify the value of expression e and where it was sourced from, e.g. $e = x + 5 \land x = \mathit{Low\_Inputs}[3]$ would say that e is the sum of the

third input obtained from a low source and the constant five. Writing invariants (the hard part of verification) in this style quickly becomes impractical, notably for advanced concepts like pointer structures. For that reason Veronica is not adequate for programs over arrays or pointers (none of their examples uses them). Lack of relational assertions also means that Veronica cannot encode declassification policies like that of the Wordle case study. From a more practical perspective, Veronica is not implemented in a dedicated auto-active verifier like our tool, Verdeca. Overall, our case studies from Section 7 are far beyond the scope what can reasonably be verified in Veronica and this assessment has been confirmed by Schoepe et al. [72] in personal communication.

Smith enforces a form of secure declassification called Qualified Release via so-called *declassification predicates* [74]. Declassification occurs via dedicated **declassify** statements, annotated by unary predicates $P(e)$ over the value e to be declassified that are evaluated in the program's *initial* state. This notion is soundly enforced in a security type system, encoded in the auto-active verifier Dafny, and applied to programs of a few SLOC each against simple policies.

Our work highlights the difficulty of proving strong constant-time guarantees for intentionally-leaky application code: such reasoning necessarily treats implementation concerns and so cannot be performed on an abstract model alone. It would be interesting therefore to extend existing constant-time programming languages [20] with support for rich security policies and declassification.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their comments and insightful suggestions that enabled us to improve this paper.

This research was sponsored by the U.S. 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.

This material is based upon work supported by the Commonwealth of Australia Defence Science and Technology Group, Next Generation Technologies Fund (NGTF)

Naumann was supported in part by NSF award CNS-1718713.

REFERENCES

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep learning with differential privacy. In *ACM CCS*. 308–318.
[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying constant-time implementations. In *USENIX Security*. 53–70.
[3] Tiago Alves and Don Felton. 2004. TrustZone: Integrated Hardware and Software Security, White Paper. *ARM, July* (2004).
[4] Miguel E. Andrés, Nicolás E. Bordenabe, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. 2013. Geo-Indistinguishability: Differential Privacy for Location-Based Systems. In *ACM CCS*. 901––914.
[5] Aslan Askarov and Stephen Chong. 2012. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *IEEE CSF*. 308–322.
[6] Aslan Askarov, Stephen Chong, and Heiko Mantel. 2015. Hybrid Monitors for Concurrent Noninterference. In *IEEE CSF*. 137–151.
[7] Aslan Askarov and Andrei Sabelfeld. 2007. Gradual Release: Unifying Declassification, Encryption and Key Release Policies. In *IEEE S&P*. 207–221.
[8] Musard Balliu, Mads Dam, and Gurvan Le Guernic. 2011. Epistemic temporal logic for information flow security. In *ACM PLAS*.
[9] Anindya Banerjee, David A Naumann, and Stan Rosenberg. 2008. Expressive declassification policies and modular static enforcement. In *IEEE S&P*. 339–353.
[10] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. Proving differential privacy via probabilistic couplings. In *LICS*. 749–758.
[11] Thomas Bauereiß, Armando Pesenti Gritti, Andrei Popescu, and Franco Raimondi. 2017. CoSMeDis: a distributed social media platform with formally verified confidentiality guarantees. In *IEEE S&P*. 729–748.
[12] Thomas Bauereiß, Armando Pesenti Gritti, Andrei Popescu, and Franco Raimondi. 2018. CoSMed: A confidentiality-verified social media platform. *J. Automated Reasoning* 61, 1 (2018), 113–139.
[13] Lennart Beringer. 2012. End-to-end Multilevel Hybrid Information Flow Control. In *Asian Symposium on Programming Languages and Systems (APLAS)*. 50–65.
[14] Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. 2015. History-based verification of functional behaviour of concurrent programs. In *SEFM 2015 Collocated Workshops*. 84–98.
[15] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX WOOT*. 11.
[16] Niklas Broberg and David Sands. 2009. Flow-sensitive semantics for dynamic information flow policies. In *ACM PLAS*. 101–112.
[17] Niklas Broberg and David Sands. 2010. Paralocks: role-based information flow control and beyond. In *POPL*, Vol. 45. 431–444.
[18] Niklas Broberg, Bart van Delft, and David Sands. 2015. The anatomy and facets of dynamic policies. In *IEEE CSF*. 122–136.
[19] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*. 249–266.
[20] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: a DSL for timing-sensitive computation. In *PLDI*. 174–189.
[21] Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Marco Stronati. 2014. A Predictive Differentially-Private Mechanism for Mobility Traces. In *PETS*. 21–41.
[22] Raj Chetty and John N Friedman. 2019. A practical method to reduce privacy loss when disclosing statistics based on small samples. In *AEA Papers and Proceedings*, Vol. 109. 414–20.
[23] Raj Chetty, John N Friedman, Nathaniel Hendren, Maggie R Jones, and Sonya R Porter. 2018. *The opportunity atlas: Mapping the childhood roots of social mobility*. Technical Report. National Bureau of Economic Research.
[24] Raj Chetty, Nathaniel Hendren, Patrick Kline, and Emmanuel Saez. 2014. Where is the land of opportunity? The geography of intergenerational mobility in the United States. *The Quarterly Journal of Economics* 129, 4 (2014), 1553–1623.
[25] Andrey Chudnov, George Kuan, and David A. Naumann. 2014. Information Flow Monitoring as Abstract Interpretation for Relational Logic. In *IEEE CSF*. 48–62.
[26] Andrey Chudnov and David A Naumann. 2018. Assuming You Know: Epistemic Semantics of Relational Annotations for Expressive Flow Policies. In *IEEE CSF*. 189–203.
[27] David Clark and Sebastian Hunt. 2008. Non-Interference for Deterministic Interactive Programs. In *Formal Aspects in Sec. and Trust (LNCS)*, Vol. 5491.
[28] David Costanzo and Zhong Shao. 2014. A separation logic for enforcing declarative information flow control policies. In *POST*. 179–198.
[29] David Costanzo, Zhong Shao, and Ronghui Gu. 2016. End-to-end verification of information-flow security for C and assembly programs. In *PLDI*. 648–664.
[30] Mads Dam. 2006. Decidability and proof systems for language-based noninterference relations. In *POPL*. 67–78.
[31] Timo M Deist, Frank JWM Dankers, Priyanka Ojha, M Scott Marshall, Tomas Janssen, Corinne Faivre-Finn, Carlotta Masciocchi, Vincenzo Valentini, Jiazhou Wang, Jiayan Chen, et al. 2020. Distributed learning on 20 000+ lung cancer patients–The Personal Health Train. *Radiotherapy and Oncology* 144 (2020), 189–200.
[32] François Dupressoir, Andrew D. Gordon, Jan Jürjens, and David A. Naumann. 2014. Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols. *J. Computer Security* 22, 5 (2014), 823–866.
[33] Cynthia Dwork. 2006. Differential Privacy. In *ICALP*. 1–12.
[34] Sebastian Eggert and Ron van der Meyden. 2017. Dynamic intransitive noninterference revisited. *Formal Aspects of Computing* 29, 6 (2017), 1087–1120.
[35] Marco Eilers, Peter Müller, and Samuel Hitz. 2018. Modular Product Programs. In *ESOP*. Springer, 502–529.
[36] Gidon Ernst, Alexander Knapp, and Toby Murray. 2022. A Hoare Logic with Regular Behavioral Specifications. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*.
[37] Gidon Ernst and Toby Murray. 2019. SecCSL: Security Concurrent Separation Logic. In *International Conference on Computer Aided Verification (CAV)*. 208–230.
[38] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3—where programs meet provers. In *ESOP*. 125–128.
[39] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. Compositional Non-Interference for Fine-Grained Concurrent Programs. In *IEEE S&P*. 1416–1433.
[40] Joseph Goguen and José Meseguer. 1982. Security Policies and Security Models. In *IEEE S&P*. 11–20.

[41] Alexey Gotsman, Josh Berdine, and Byron Cook. 2011. Precision and the conjunction rule in concurrent separation logic. *ENTCS* 276 (2011), 171–190.

[42] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. 2018. Chiron: Privacy-preserving machine learning as a service. *arXiv preprint arXiv:1803.05961* (2018).

[43] Nick Hynes, Raymond Cheng, and Dawn Song. 2018. Efficient Deep Learning on Multi-Source Private Data. arXiv:1807.06689 [cs.LG]

[44] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*. Springer, 41–55.

[45] Bart Jacobs, Frédéric Vogels, and Frank Piessens. 2015. Featherweight verifast. *Logical Methods in Computer Science* 11 (2015).

[46] Aleksandr Karbyshev, Kasper Svendsen, Aslan Askarov, and Lars Birkedal. 2018. Compositional Non-Interference for Concurrent Programs via Separation and Framing. In *POST*.

[47] David Kohlbrenner and Hovav Shacham. 2017. On the effectiveness of mitigations against floating-point timing channels. In *USENIX Security*. 69–81.

[48] Roland Kunkel, Do Le Quoc, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. 2019. TensorSCONE: A Secure TensorFlow Framework using Intel SGX. arXiv:1902.04413 [cs.CR]

[49] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*. 557–574.

[50] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *LPAR*. 348–370.

[51] K Rustan M Leino and Michał Moskal. 2010. Usable auto-active verification. In *Usable Verification Workshop. http://fm.csl.sri.com/UV10*.

[52] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. A Secure and Formally Verified Linux KVM Hypervisor. In *IEEE S&P*.

[53] Heiko Mantel, David Sands, and Henning Sudbrock. 2011. Assumptions and Guarantees for Compositional Noninterference. In *IEEE CSF*. 218–232.

[54] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *HASP*. Article 10, 1 pages.

[55] Fan Mo, Hamed Haddadi, Kleomenis Katevas, Eduard Marin, Diego Perino, and Nicolas Kourtellis. 2021. PPFL: privacy-preserving federated learning with trusted execution environments. *arXiv preprint arXiv:2104.14380* (2021).

[56] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. Cachezoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 69–90.

[57] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. 2020. CopyCat: Controlled Instruction-Level Attacks on Enclaves. In *USENIX Security*. 469–486.

[58] Peter Müller, Malte Schwerhoff, and Alexander J Summers. 2016. Viper: A verification infrastructure for permission-based reasoning. In *VMCAI*. 41–62.

[59] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: from General Purpose to a Proof of Information Flow Enforcement. In *IEEE S&P*. 415–429.

[60] Toby Murray, Robert Sison, and Kai Engelhardt. 2018. COVERN: A Logic for Compositional Verification of Information Flow Control. In *EuroS&P*.

[61] Toby Murray and Paul C. van Oorschot. 2018. BP: Formal Proofs, the Fine Print and Side Effects. In *IEEE Cybersecurity Development Conference (SecDev)*. IEEE.

[62] Peter W O'Hearn. 2004. Resources, concurrency and local reasoning. In *International Conference on Concurrency Theory (CONCUR)*. Springer, 49–67.

[63] Gaurav Parthasarathy, Peter Müller, and Alexander J Summers. 2021. Formally validating a practical verification condition generator. In *International Conference on Computer Aided Verification (CAV)*. 704–727.

[64] Willem Penninckx, Amin Timany, and Bart Jacobs. 2019. Specifying I/O using abstract nested Hoare triples in separation logic. In *FTfJP*. 1–7.

[65] Andrei Popescu, Thomas Bauereiss, and Peter Lammich. 2021. Bounded-deducibility security. In *ITP*.

[66] Andrei Popescu, Peter Lammich, and Ping Hou. 2021. CoCon: A conference management system with formally verified document confidentiality. *J. Automated Reasoning* 65, 2 (2021), 321–356.

[67] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Capkun. 2020. Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend. In *USENIX Security*. 663–680.

[68] John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *LICS*. 55–74.

[69] Alejandro Russo. 2008. *Language Support for Controlling Timing-Based Covert Channels*. Ph.D. Dissertation. Chalmers University of Technology.

[70] Andrei Sabelfeld and David Sands. 2000. Probabilistic Noninterference for Multi-Threaded Programs. In *IEEE CSFW*. 200–214.

[71] Andrei Sabelfeld and David Sands. 2009. Declassification: Dimensions and principles. *J. Computer Security* 17, 5 (2009), 517–548.

[72] Daniel Schoepe, Toby Murray, and Andrei Sabelfeld. 2020. VERONICA: Expressive and Precise Concurrent Information Flow Security. In *IEEE CSF*. 79–94.

[73] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2022. Enforcing Fine-grained Constant-time Policies. In *ACM CCS*. 83–96.

[74] Graeme Smith. 2022. Declassification Predicates for Controlled Information Release. In *ICFEM*. 298–315.

[75] Florian Tramèr and Dan Boneh. 2019. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. arXiv preprint arXiv:1806.03287. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1806.03287

[76] Viktor Vafeiadis. 2011. Concurrent Separation Logic and Operational Semantics. In *Mathematical Foundations of Programming Semantics (MFPS)*. 335–351.

[77] Bart van Delft, Sebastian Hunt, and David Sands. 2015. Very static enforcement of dynamic policies. In *POST*. 32–52.

[78] Kang Wei, Jun Li, Ming Ding, Chuan Ma, Howard H Yang, Farhad Farokhi, Shi Jin, Tony QS Quek, and H Vincent Poor. 2020. Federated learning with differential privacy: Algorithms and performance analysis. *IEEE Transactions on Information Forensics and Security* 15 (2020), 3454–3469.

[79] Pengbo Yan and Toby Murray. 2021. SecRSL: Security Separation Logic for C11 Release-Acquire Concurrency. *Proc. ACM Program. Lang.* 5 (OOPSLA), 99 (2021).

[80] Jean Yang. 2015. *Preventing information leaks with policy-agnostic programming*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[81] Chenyi Zhang. 2011. Conditional information flow policies and unwinding relations. In *Int. Symp. on Trustworthy Global Computing (TGC)*. 227–241.

A PROOF OF THE MOTIVATING EXAMPLE

Recall (Section 2) that the example of Fig. 1 we verify by defining a resource invariant that links the input/output history tr of the program and its state, to which the pointer `struct avg_state * st` points:

$$\begin{aligned}\mathsf{inv}(tr, \mathtt{st}) \quad \hat{=} \quad & \mathcal{H}(tr) \wedge \mathtt{st\text{->}count} = \mathsf{length}(tr) \\ & \wedge \mathtt{st\text{->}sum} = \mathsf{sum}(tr)\end{aligned}$$

Recall that we attach this resource invariant to the lock, which is implemented by the `avg_lock()` and `avg_unlock()` functions that, respectively, soundly acquire and release this invariant [37] by ensuring that it always holds whenever the shared state `st` is accessed. This we specify via standard annotations as follows (recalling that **`result`** in a postcondition refers to the function's return value):

```
struct avg_state * avg_lock();
_(ensures inv(tr, result))

void avg_unlock(struct avg_state *st);
_(requires inv(tr, st))
```

Recall that the declassification policy $\mathcal{D}(tr)$ for this example is:

$$\mathcal{D}(tr) \quad \hat{=} \quad \mathsf{length}(tr) \geq 6 \rightsquigarrow \mathsf{sum}(tr)/\mathsf{length}(tr) :: \mathbf{low}$$

The average of the inputs can be declassified so long as there are at least 6 inputs.

Then the proof appears in Fig. 9. The proof is carried out using the rules of our logic (Section 4), automated by Verdeca; intermediate proof states we annotate in purple, so the reader can see how the proof progresses. Doing so proves the policy-agnostic security guarantee: the program leaks no more information than that contained in assume statements. To prove all leakage is in accordance with the declassification policy, we apply the rules of Section 6 to collect audit obligations. In this case, we need to prove that the underlined path condition in Fig. 9 (what is known at the time of the assumption) is sufficient to justify the assumption against the declassification policy. Specifically, letting P be the underlined path condition, we have to check (Definition 6.2) that P implies the policy condition $\mathsf{length}(tr) \geq 6$ and that P and the policy release formula $\mathsf{sum}(tr)/\mathsf{length}(tr) :: \mathbf{low}$ together imply the assumption `avg` :: **`low`**. These trivially hold, thus the program satisfies the secure declassification against this declassification policy: the program leaks no more information than that allowed by its declassification policy (by Theorem 6.5).

In practice, (see Section 8) we often inline the check of the audit obligation (Definition 6.2) into the proof, so it can be automatically discharged by Verdeca. This can be done for instance by replacing the line `_(assume avg :: low)` in Fig. 9 with the following three:

```
_(assert length(tr) ≥ 6) // check that P ⟹ φ_D
_(assume sum(tr)/length(tr) :: low) // assume, therefore ρ_D
_(assert avg :: low) // check that P ⋆ ρ_D ⟹ ρ
```

We explain this transformation for an arbitrary declassification policy $\varphi_{\mathcal{D}} \rightsquigarrow \rho_{\mathcal{D}}$ whose condition is $\varphi_{\mathcal{D}}$ and release formula is $\rho_{\mathcal{D}}$, and for assumption `_(assume` ρ`)`, as indicated in the comments. The first line checks that the policy condition $\varphi_{\mathcal{D}}$ holds, under the current path condition (called P in Fig. 9). This is the first check of Definition 6.2. Having proved that $\varphi_{\mathcal{D}}$ holds, the second line then makes use of the policy release formula $\rho_{\mathcal{D}}$. The path condition after the second line is therefore $P \star \rho_{\mathcal{D}}$. Thus the fourth line then checks that the original assumption (in this case `avg` :: **`low`**) holds, i.e., writing ρ for this assumption, that $P \star \rho_{\mathcal{D}} \implies \rho$, the second check of Definition 6.2.

B PROOFS OF THE MAIN THEOREMS

The proofs of the main theorems Theorem 5.6 and Theorem 6.5 are expressed with respect to an inductive generalization that captures all necessary properties of two executions running in lockstep. The respective soundness proofs will therefore rely on an intermediate result that precisely characterizes how a major run is related to any minor run.

Definition B.1 (Aligned actions and schedules). Two actions a and a' are *aligned* wrt. a security level ℓ, written $a \cong_\ell a'$, if one of the listed cases applies. Two schedules are aligned, written $\sigma \cong_\ell \sigma'$, if they have the same length and their actions are point-wise aligned.

$$\begin{array}{cccc} \tau \cong_\ell \tau & \mathbf{L} \cong_\ell \mathbf{L} & \mathbf{R} \cong_\ell \mathbf{R} & \mathbf{Load}\ p \cong_\ell \mathbf{Load}\ p \\ \mathbf{Assm}\ s\ \rho \cong_\ell \mathbf{Assm}\ s'\ \rho & \mathbf{Trace}\ e \cong_\ell \mathbf{Trace}\ e' & & \mathbf{Store}\ p \cong_\ell \mathbf{Store}\ p \\ \mathbf{Out}\ \ell'\ v \cong_\ell \mathbf{Out}\ \ell'\ v' & \text{if } \ell \sqsubseteq \ell' \implies v = v' & & \end{array}$$

Aligned schedules capture that the type of events and formula ρ for assumptions matches per step, and that an attacker cannot learn information form memory access and from outputs (equality of pointers p as well as values v, v'). Note that the condition is strictly stronger than observably equivalent schedules $\sigma \approx_\ell \sigma'$ (Definition 5.2), specifically, it enforces that the event type is always the same even for unobservable events and that assumption steps are paired with the *same* assumed formula ρ.

Soundness is characterized with the help of an inductive predicate $\mathsf{secure}^n_\ell(P_1, tr_1, c, Q, tr, A)$ which states that the program is safe to execute, correct, and secure for n steps similar to [37, Def. 3] for SecCSL and its non-relational counterpart from [76] for standard CSL. In comparison to [37] it *additionally* tracks alignment between possible schedules of the execution of c, injects assumption steps into intermediate path conditions, tracks the history trace of events, and collects audit triples in A. As such, it encodes all consequences of extended judgements $\vdash_\ell \{P_1 \star \mathcal{H}(tr_1)\}\ c\ \{Q \star \mathcal{H}(tr)\} \triangleright A$ to prove Theorems 5.6 and 6.5 but while this judgement is defined compositionally over the *structure of the program command* c, predicate $\mathsf{secure}^n_\ell(P_1, tr_1, c, Q, tr, A)$ unwinds individual *execution steps* linearly. This is the key gap that is bridged in the soundness proof.

Definition B.2 (Secure Executions). Predicate secure is defined recursively over the counter n of remaining steps to assert secure:

- $\mathsf{secure}^0_\ell(P_1, tr_1, c, Q, tr, A)$ holds always, i.e., a program is secure for zero steps.
- $\mathsf{secure}^{n+1}_\ell(P_1, tr_1, c, Q, tr, A)$ holds, if for all possible pairs of first steps $(\mathbf{run}\ L_1, c_1, s_1, h_1) \xrightarrow{\sigma_1} k_2$ and $(\mathbf{run}\ L, c_1, s_1', h_1') \xrightarrow{\sigma_1'} k_2'$ starting from states $(s_1, h_1), (s_1', h_1') \models P_1 \star \mathcal{H}(tr_1) \star \mathsf{invs}(L_1)$ we have

(1) $\sigma_1 \cong_\ell \sigma_1'$ are aligned according to Definition B.1, and

```
void avg_sum_thread() {
  while(true) {
    struct avg_state * st = avg_lock();
    {inv(tr, st)}
    {H(tr) ∧ st->count = length(tr) ∧ st->sum = sum(tr)}
    int i = avg_get_input();
    {H(tr · i) ∧ st->count = length(tr) ∧ st->sum = sum(tr)}
    st->count += 1;
    {H(tr · i) ∧ st->count = length(tr · i) ∧ st->sum = sum(tr)}
    st->sum += i;
    {H(tr · i) ∧ st->count = length(tr · i) ∧ st->sum = sum(tr · i)}
    {inv(tr · i, st)}
    avg_unlock(st);
  }
}

void avg_declass_thread() {
  struct avg_state * st = avg_lock();
  {inv(tr, st)}
  {H(tr) ∧ st->count = length(tr) ∧ st->sum = sum(tr)}
  if (st->count >= 6) {
    {H(tr) ∧ st->count = length(tr) ∧ st->sum = sum(tr) ∧ sum(tr) ≥ 6}
    int avg = st->sum / st->count;
    {H(tr) ∧ st->count = length(tr) ∧ st->sum = sum(tr) ∧ sum(tr) ≥ 6 ∧ avg = sum(tr)/length(tr)}
    _(assume avg :: low)                                                    ▷{(P, tr, avg :: low)}
    {H(tr) ∧ st->count = length(tr) ∧ st->sum = sum(tr) ∧ sum(tr) ≥ 6 ∧ avg = sum(tr)/length(tr) ∧ avg :: low}
    print_average(avg);
    {H(tr) ∧ st->count = length(tr) ∧ st->sum = sum(tr) ∧ sum(tr) ≥ 6 ∧ avg = sum(tr)/length(tr) ∧ avg :: low}
  }
  {H(tr) ∧ st->count = length(tr) ∧ st->sum = sum(tr)}
  {inv(tr, st)}
  avg_unlock(st);
}
```

Figure 9: Proof of the example in Fig. 1. The use of the assume statement induces an audit triple $(P, tr, \texttt{avg :: low})$, where P is the underlined path condition at the point of the assume statement, namely $\texttt{st->count} = \mathsf{length}(tr) \wedge \texttt{st->sum} = \mathsf{sum}(tr) \wedge \mathsf{sum}(tr) \geq 6 \wedge \texttt{avg} = \mathsf{sum}(tr)/\mathsf{length}(tr)$. The declassification policy $\varphi_{\mathcal{D}} \rightsquigarrow \rho_{\mathcal{D}}$ is honored (Definition 6.2) if P implies $\varphi_{\mathcal{D}}$ and and $P \star \rho_{\mathcal{D}}$ implies the assumption avg :: low. The policy for this example recall is $\varphi_{\mathcal{D}}(tr) \mathrel{\hat{=}} \mathsf{length}(tr) \geq 6$, and $\rho_{\mathcal{D}}(tr) \mathrel{\hat{=}} \mathsf{sum}(tr)/\mathsf{length}(tr)$ :: low. P clearly implies $\varphi_{\mathcal{D}}$; moreover, so does $P \star \rho_{\mathcal{D}}$ imply avg :: low. Thus the example is secure against the declassification policy, by Theorem 6.5.

(2) the two configurations k_2 and k_2' are *matched*, in the sense that either both are **stop**ped with the same lock-set L_2 or both are **run**ning with identical commands c_2 and lock-set L_2, and
(3) if the execution step was an assumption ρ, A must contain a corresponding audit triple (P_1', tr_1, ρ) for current trace tr_1 and some assertion P_1' that follows from the current path condition P_1, i.e., $P_1 \implies P_1'$ (weakening is allowed and necessary to validate the consequence rule), and
(4) if $k_2 = (\textbf{stop}\ L_2, s_2, h_2)$ and $k_2' = (\textbf{stop}\ L_2, s_2', h_2')$ then either the step was a violated assumption ρ with $s_1, s_1' \not\models \rho$ or the postcondition holds $(s_1, h_1), (s_1', h_1') \models Q \star \mathcal{H}(tr_2) \star \mathsf{invs}(L_2)$ for some tr_2, and
(5) if $k_2 = (\textbf{run}\ c_2, L_2, s_2, h_2)$ and $k_2' = (\textbf{run}\ c_2, L_2, s_2', h_2')$ then either the step was a violated assumption ρ with $s_1, s_1' \not\models \rho$ or some intermediate assertion P_2 holds $(s_1, h_1), (s_1', h_1') \models P_2 \star \mathcal{H}(tr_2) \star \mathsf{invs}(L_2)$ for some tr_2 as well as recursively, the program is secure for the remaining n steps from that point on, i.e., $\mathsf{secure}_\ell^n(P_2, tr_2, c_2, Q, tr, A)$.

So far, we have tacitly suppressed the semantic model underlying the abstract predicate $\mathcal{H}(tr)$. It can be explained either by introducing a constant ghost location tr in heaps, so that $(s, h), (s', h') \models \mathcal{H}(tr)$ iff $h = [\mathrm{tr} \mapsto [\![tr]\!]_s]$ and $h' = [\mathrm{tr} \mapsto [\![tr]\!]_{s'}]$ (recall that tr is an expression and cf. [9, 72]), or alternatively we can interpret tr in terms of yet another type of actions in the schedule (which we have done in our Isabelle/HOL proofs).

Lemma B.3. *A valid proof using the rules* $\vdash_\ell \{P_1 \star \mathcal{H}(tr_1)\}\ c\ \{Q \star \mathcal{H}(tr)\} \triangleright A$, *implies that the program is secure for any number of steps, i.e.,* $\forall n.\ \mathsf{secure}_\ell^n(P_1, tr_1, c, Q, tr, A)$.

Proof. By induction on the derivation of $\vdash_\ell \{P_1 \star \mathcal{H}(tr_1)\}\ c\ \{Q \star \mathcal{H}(tr)\} \triangleright A$. Structural rules (frame, conseq) and compound statements (if, while, sequential and parallel composition) need an inner induction on the number of steps n. In our mechanized development, each case is formulated as a separate lemma. □

Lemma B.4 (Secure, Lock-step runs). *Assume for all n, that* $\mathsf{secure}^n_\ell(P_1, tr_1, c, Q, tr, A)$. *For a major run* $(\mathbf{run}\ L, c, s, h) \xrightarrow{\sigma} k$ *and a minor run* $(\mathbf{run}\ L, c, s', h') \xrightarrow{\sigma'} k'$ *with the* same *program c, lockset L and $|\sigma| = |\sigma'|$ such that $(s, h), (s', h') \models_\ell P_1 \star \mathcal{H}(tr_1)\mathsf{invs}(L)$, we have*

- *the two configurations k and k' are* matched, *in the sense that either both are **stop**ped with the same lock-set L_2 or both are **run**ning with identical commands c_2 and lock-set L_2,*

and one of the following is true:

- *an assumption has failed at some step $m < |\sigma|$, i.e., predicate* $\mathsf{assumption\text{-}failed}_\ell(m, \sigma, \sigma')$ *holds and the two prefix runs given as* $(\mathbf{run}\ L, c, s, h) \xrightarrow{\sigma_{|m+1}} k_{m+1}$ *and* $(\mathbf{run}\ L, c, s', h') \xrightarrow{\sigma'_{|m+1}} k'_{m+1}$ *are characterized as follows*
  - *(1) the intermediate configurations k_{m+1} and k'_{m+1} are matched (cf. above), and*
  - *(2) the schedules are aligned up to and including that step, and $\sigma_{|m+1} \cong_\ell \sigma'_{|m+1}$, and*
  - *(3) there is an intermediate assertion P_k and trace expression tr_k so that $(P_k, tr_k, \rho) \in A$, where $\sigma(k) = \sigma'(k) = \mathbf{Assm}\ \rho$ is the failed assertion, and $P_k \star \mathcal{H}(tr_k) \star \mathsf{invs}(L_k)$ holds in the states of k_{m+1} and k'_{m+1} for the corresponding lock-set L_k.*
- *no assumption has failed and the entire runs are aligned $\sigma \cong_\ell \sigma'$, where k and k' either both stopped and validate postcondition $Q \star \mathcal{H}(tr') \star \mathsf{invs}(L')$ for the some trace expression tr' and respective lock-set L' of k and k', or they are both running with the same residual program c' and similarly validate some intermediate assertion P' and trace expression tr' from which c' is again secure for any number of steps,* $\forall n.\ \mathsf{secure}^n_\ell(P', tr', c', Q, tr, A)$.

Sketch. This lemma is proved by induction on the (locked) steps of the two executions, unfolding the inductive security property alongside. □

Lemma B.5. *Consider a verified program $\vdash_\ell \{P_1 \star \mathcal{H}(tr_1)\}\ c\ \{Q \star \mathcal{H}(tr')\} \triangleright A$, i.e., $\forall n.\ \mathsf{secure}^n_\ell(P_1, tr_1, c, Q, tr, A)$, a policy D that has been formally audited (Definition 6.2) and a pair of a major run $(\mathbf{run}\ L_1, c, s_1, h_1) \xrightarrow{\sigma_1}{}^* k_1$ and minor run $(\mathbf{run}\ L_1, c, s'_1, h'_1) \xrightarrow{\sigma'_1}{}^* k'_1$ from the precondition $(s_1, h_1), (s'_1, h'_1) \models P_1 \star \mathsf{invs}(L_1)$. Then each assumption failure at some $m < |\sigma| = |\sigma'|$ with $\sigma(m) = \mathbf{Assm}\ s_m\ \rho$, $\sigma'(m) = \mathbf{Assm}\ s'_m\ \rho$ is paired with an entry with $(P_m, tr_m, \rho) \in A$ and intermediate states $L_m, s_m, h_m, s'_m, h'_m$ with $(s_m, h_m), (s'_m, h'_m) \models P_m \star \mathsf{invs}(L_m)$, such that the specified traces match the schedule: $[\![tr_m]\!]_{s_m} = [\![tr_1]\!]_{s_1} \cdot \mathsf{trace}(\sigma_{|m})$ and $[\![tr_m]\!]_{s'_m} = [\![tr_1]\!]_{s'_1} \cdot \mathsf{trace}(\sigma'_{|m})$.*

Sketch. This lemma is proved by induction on the (locked) steps of the two executions, unfolding the inductive security property alongside. □

Theorem 5.6 (Policy-agnostic security guarantee). *If $\vdash_\ell \{P\}\ c\ \{Q\}$ then for a major run $(\mathbf{run}\ L, c, s, h) \xrightarrow{\sigma_1}{}^* k_1$ the knowledge gain from one additional step $k_1 \xrightarrow{\sigma_2} k_2$, expressed as the difference in uncertainty, is bounded by the release condition:*

$$\mathsf{uncertainty}_\ell(P, \sigma_1, c, L, s, h) \setminus \mathsf{uncertainty}_\ell(P, \sigma_1 \cdot \sigma_2, c, L, s, h) \subseteq \mathsf{assumed\text{-}release}_\ell(P, \sigma_1, c, L, s, h)$$

Sketch. Unfolding the definitions, we obtain a minor run in the set difference $(\mathbf{run}\ L, c, s', h') \xrightarrow{\sigma'_1}{}^* k'_1$ that is still uncertain, i.e., with $\sigma_1 \simeq_\ell \sigma'_1$, but none of its extensions are. Considering the two cases from Lemma B.4, noting that $k_1 = (\mathbf{run}\ L, c_1, s_1, h_1)$ must be running and $\vdash_\ell \{P_1\}\ c_1\ \{Q\}$ for some P_1.

- If this pair of runs already contains a failed assumption, then it witnesses the release condition, even if the attacker has not been able to observe any consequence of that fact yet.
- Otherwise, since k_1 produces another step we have a matching $k'_1 \xrightarrow{\sigma_2} k'_2$ (both k_1 and k'_1 are running and the small-step semantics is left-total), and this extension leaks information, i.e., $\sigma_2 \not\simeq_\ell \sigma'_2$. Applying Lemma B.4 again from P_1 for just that step produces a contradiction: because we have the stronger condition $\sigma_2 \cong_\ell \sigma'_2$ from assumption steps (which are invisible) *and* for regular steps (which are proven secure).

□

Theorem 6.5 (Policy-specific security guarantee). *For a verified program $\vdash_\ell \{P \star \mathcal{H}(\langle\rangle)\}\ c\ \{Q \star \mathcal{H}(tr')\} \triangleright A$ and a policy D formally audited according to Definition 6.2 for each major run $(\mathbf{run}\ L, c, s, h) \xrightarrow{\sigma_1}{}^* k_1$ with final step $k_1 \xrightarrow{\sigma_2} k_2$:*

$$\mathsf{assumed\text{-}release}_\ell(P, \sigma, c, L, s, h) \subseteq \mathsf{policy\text{-}release}_\ell(\mathcal{D}, P, \sigma, c, L, s, h)$$

Sketch. Fix a minor run with schedule σ' from assumed-release that has an assumption failure at step with respect to the major run. By Lemma B.4 this occurs at some point n up to which $\sigma_{1|n} \cong \sigma'_{1|n}$ which is critical for some side-conditions. By Lemma B.5 for $tr_1 = \langle\rangle$, $P_1 = P$, and the runs up to step n we obtain an corresponding audit triple (P_2, tr', ρ) for which Definition 6.2 guarantees $\varphi_{\mathcal{D}}$ is implied by that P_2, but the failed assumption ρ falsifies $\rho_{\mathcal{D}}$ (contraposition of the second implication of the audit), and therefore Definition 6.3 is satisfied. □

C PROGRAM SEMANTICS

The single-step operational semantics is defined in Fig. 10; multiple steps of execution $k \xrightarrow{\sigma}{}^* k'$ is defined inductively below. The assertion semantics are defined in Fig. 3.

$$\frac{}{k \xrightarrow{\langle\rangle}{}^* k} \qquad \frac{k \xrightarrow{\sigma_1} k_1 \qquad k_1 \xrightarrow{\sigma_2}{}^* k_2}{k \xrightarrow{(\sigma_1 \cdot \sigma_2)}{}^* k_2}$$

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

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

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

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

$$\frac{(\textbf{run}\ c_1, L, s, h) \xrightarrow{\sigma} (\textbf{run}\ c_1', L', s', h')}{(\textbf{run}\ c_1 \parallel c_2, L, s, h) \xrightarrow{\langle \textbf{L} \rangle \cdot \sigma} (\textbf{run}\ c_1' \parallel c_2, L', s', h')} \qquad \frac{\text{if } s \vDash e \text{ then } c' = c_1 \text{ else } c' = c_2}{(\textbf{run if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2, L, s, h) \xrightarrow{\langle \tau \rangle} (\textbf{run}\ c', L, s, h)}$$

$$\frac{s \nvDash e}{(\textbf{run while}\ e\ \textbf{do}\ c, L, s, h) \xrightarrow{\langle \tau \rangle} (\textbf{stop}\ L, s, h)} \qquad \frac{s \vDash e}{(\textbf{run while}\ e\ \textbf{do}\ c, L, s, h) \xrightarrow{\langle \tau \rangle} (\textbf{run}\ c; \textbf{while}\ e\ \textbf{do}\ c, L, s, h)}$$

$$\frac{}{(\textbf{run skip}, L, s, h) \xrightarrow{\langle \tau \rangle} (\textbf{stop}\ L, s, h)} \qquad \frac{}{(\textbf{run assume}\ \rho, L, s, h) \xrightarrow{\langle \textbf{Assm}\ \rho \rangle} (\textbf{stop}\ L, s, h)}$$

$$\frac{}{(\textbf{run output}\ \ell'\ e_v, L, s, h) \xrightarrow{\langle \textbf{Out}\ [\![\ell']\!]_s\ [\![e_v]\!]_s \rangle} (\textbf{stop}\ L, s, h)} \qquad \frac{}{(\textbf{run trace}\ e, L, s, h) \xrightarrow{\langle \textbf{Trace}\ [\![e]\!]_s \rangle} (\textbf{stop}\ L, s, h)}$$

Figure 10: Small-step operational semantics. Symmetric parallel rules in which c_2 is scheduled producing the event R have been omitted in the interests of brevity. We write $s \vDash e$ when evaluating expression e in state s and casting the resulting value to a boolean yields the value `true`; we write $s \nvDash e$ otherwise.