VERONICA: Expressive and Precise Concurrent Information Flow Security

Daniel Schoepe*, Toby Murray† and Andrei Sabelfeld*
*Chalmers University of Technology †University of Melbourne and Data61

***Abstract*—Methods for proving that concurrent software does not leak its secrets has remained an active topic of research for at least the past four decades. Despite an impressive array of work, the present situation remains highly unsatisfactory. With contemporary compositional proof methods one is forced to choose between *expressiveness* (the ability to reason about a wide variety of security policies), on the one hand, and *precision* (the ability to reason about complex thread interactions and program behaviours), on the other. Achieving both is essential and, we argue, requires a new style of compositional reasoning.**

We present VERONICA, the first program logic for proving concurrent programs information flow secure that supports compositional, high-precision reasoning about a wide range of security policies and program behaviours (e.g. expressive declassification, value-dependent classification, secret-dependent branching). Just as importantly, VERONICA embodies a new approach for engineering such logics that can be re-used elsewhere, called *decoupled functional correctness* (DFC). DFC leads to a simple and clean logic, even while achieving this unprecedented combination of features. We demonstrate the virtues and versatility of VERONICA by verifying a range of example programs, beyond the reach of prior methods.

1. Introduction

Software guards our most precious secrets. More often than not, software systems are built as a collection of concurrently executing threads of execution that cooperate to process data. In doing so, these threads collectively implement security policies in which the sensitivity of the data being processed is often data-dependent [1]–[10], and the rules about to whom it can be disclosed and under what conditions can be non-trivial [11]–[16]. The presence of concurrency greatly complicates reasoning, since a thread that behaves securely when run in isolation can be woefully insecure in the presence of interference from others [10], [17]–[19] or due to scheduling [20], [21].

For these reasons, being able to formally prove that concurrent software does not leak its secrets (to the wrong places at the wrong times) has been an active and open topic of research for at least the past four decades [22], [23]. Despite an impressive array of work over that time, the present situation remains highly unsatisfactory. With contemporary proof methods one is forced to choose between *expressiveness* (e.g. [24]–[27]), on the one hand, and *precision* (e.g. [10], [19], [28]–[33]), on the other.

By expressiveness, we mean the ability to reason about the enforcement of a wide variety of security policies and classes thereof, such as state-dependent secure declassification and data-dependent sensitivity. It is well established that, beyond simple noninterference [34] ("secret data should never be revealed in public outputs"), there is no universal solution to specifying information flow policies [13], and that different applications might have different interpretations on what adherence to a particular policy means.

By precision, we mean the ability to reason about complex thread interactions and program behaviours. This includes not just program behaviours like secret-dependent branching that are beyond the scope of many existing proof methods (e.g. [10], [19], [28]). Moreover, precision is aided by reasoning about each thread under local assumptions that it makes about the behaviour of the others [30], [35]. For instance [10], suppose thread B receives data from thread A, by acquiring a lock on a shared buffer and then checking the buffer contents. Thread B relies on thread A having appropriately labelled the buffer to indicate the (data-dependent) sensitivity of the data it contains and, while thread B holds the lock, it relies on all other threads to avoid modifying the buffer (to preserve the correctness of the sensitivity label). Precise reasoning here should take account of these kinds of assumptions when reasoning about thread B and, correspondingly, should prove that they are adhered to when reasoning about thread A.

Besides expressiveness and precision, another useful property for a proof method to have is *compositionality*. We say that a proof method is *compositional* [36]–[39] when it can be used to establish the security of the entire concurrent program by using it to prove each thread secure separately.

So far it has remained an open problem of how to design a proof method (e.g. a security type system [40] or program logic [41]) that is (a) compositional, (b) supports proving a general enough definition of security to encode a variety of security policies, and (c) supports precise reasoning. We argue that achieving all three together requires a new style of program logic for information flow security.

In this paper, we present VERONICA. VERONICA is, to our knowledge, the first compositional program logic for proving concurrent programs information flow secure that supports high-precision reasoning about a wide range of security policies and program behaviours (e.g. expressive declassification, value-dependent classification, secret-dependent branching). Just as importantly, VERONICA embodies a new approach for engineering such logics that can be re-used elsewhere. This approach we call *decoupled functional correctness* (DFC), which we have found leads to a simple and clean logic, even while achieving this unprecedented combination of features. Precision is supported by reasoning about a program's functional properties. However, the key insight of DFC is that this reasoning can and should be separated from reasoning about its information flow security. As we explain, DFC exploits compositional functional correctness as a common means to unify together reasoning about various security concerns.

We provide an overview of VERONICA in Section 2. Section 3 then describes the general security property that it enforces, and so formally defines the threat model. Section 4 describes the programming language over which VERONICA has been developed. Section 5 then describes the VERONICA logic, whose virtues are further demonstrated in Section 6. Section 7 considers related work before Section 8 concludes.

All results in this paper have been mechanised in the interactive theorem prover Isabelle/HOL [42]. Our Isabelle formalisation is available online [43]. Some technical material has been omitted for brevity and can be found in the extended version of this paper [44].

2. An Overview of VERONICA

2.1. Decoupling Functional Correctness

Figure 1a depicts the data-flow architecture for a very simple, yet illustrative, example system. This example is inspired by a real world security-critical shared-memory concurrent program [10]. This example purposefully avoids some of VERONICA's features (e.g. secret-dependent branching and runtime state-dependent declassification policies), which we will meet later in Section 6. Verifying it requires highly precise reasoning, and the security policy it enforces involves both data-dependent sensitivity and delimited release style declassification [45], features that until now have never been reconciled before.

The system comprises four threads, whose code appears in Figure 1 (simplified a little for presentation). The four threads make use of a shared buffer buf protected by a lock ℓ, which also protects the shared flag variable $valid$. The top-middle thread (Figure 1c) copies data into the shared buffer, from one of two input/output (IO) *channels*: $\bot$ (a public channel whose contents is visible to the attacker) and $\top$ (a private channel, not visible to the attacker). The right-top thread (Figure 1e) reads data from the shared buffer buf and copies it to an appropriate output buffer (either $\bot buf$ for $\bot$ data or $\top buf$ for $\top$ data) for further processing by the remaining two output threads.

Each of the bottom threads outputs from its respective output buffer to its respective channel; one for $\top$ data (Figure 1b) and the other for $\bot$ data (Figure 1d).

The decision of the top-middle thread (Figure 1c, line 2), whether to input from the $\bot$ channel or the $\top$ one, is dictated by the shared variable $inmode$. The $valid$ variable (initially zero) is set to 1 by the top-middle thread once it has filled the buf variable, and is then tested by the top-right thread (Figure 1e, line 2) to ensure it doesn't consume data from buf before the top-middle thread has written to buf.

The top-right thread's decision (Figure 1e, line 3) about which output buffer it should copy the data in buf to is dictated by the $outmode$ variable. When $outmode$ indicates that the $\top$ buffer $\top buf$ should be used, the top-right thread additionally performs a signature check (via the CK function, lines 7–8) on the data to decide if it is safe to declassify and copy additionally to the $\bot buf$ output buffer. This concurrent program implements a *delimited release* [45] style declassification policy, which states that $\top$ data that passes the signature check, plus the results of the signature check itself for all $\top$ data, are safe to declassify. The language of VERONICA includes the *declassifying assignment* command $\widehat{:=}$ and the *declassifying output* command $\widehat{!}$. Besides delimited release style declassification policies, we we will see later in the examples of Section 6 that our security condition also supports stateful declassification policies.

Clearly, if $inmode$ and $outmode$ disagree, the concurrent execution of the threads might behave insecurely (e.g. the top-middle thread might place private $\top$ data into buf, which the top-right thread then copies to $\bot buf$ and is subsequently output on the public channel $\bot$). Therefore, the security of this concurrent program rests on the shared data *invariant* that $inmode$ and $outmode$ agree (whenever lock ℓ is acquired and released). This is a functional correctness property. There are a number of other such functional properties, somewhat more implicit, on which the system's security relies, e.g. that neither thread will modify buf unless they hold the lock ℓ, and likewise for $inmode$ and $outmode$, plus that only one thread can hold the lock ℓ at a time.

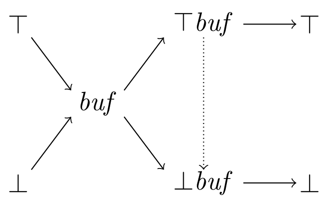
Similarly, the security of the declassification actions performed by the top-right thread rests on the fact that it only declassifies after successfully performing the signature check, in accordance with the delimited release policy.

Thus one cannot reason about the security of this concurrent program in the absence of functional correctness. However, one of the fundamental insights of VERONICA is that functional correctness reasoning should be *decoupled* from security reasoning. This is in contrast to many recent logics for concurrent information flow security, notably [28], the COVERN logic of [10] and its antecedents [19], [30] as well as [31], [32] plus many prior logics for sequential programs [2], [5], [8], [9], [46], [47] and hardware designs [48].

VERONICA decouples functional correctness reasoning from security reasoning by performing the latter over programs that carry *functional correctness annotations* $\{A_i\}$ on each program statement s_i. Thus program statements are of the form $\{A_i\}\ s_i$. Here, $\{A_i\}$ should be thought of as akin to a Hoare logic precondition [49]. It states conditions that are known to be true whenever statement s_i is executed in the *concurrent* program. We call this resulting approach *decoupled functional correctness* (DFC).

The contents of each of the annotations in Figure 1c and Figure 1e have been omitted in the interests of brevity (they can be found in our Isabelle formalisation), and simply replaced by identifiers $\{A_i\}$.

For verifying the security of the top-right thread (Figure 1e), annotations $\{A_{11}\}$ through $\{A_{15}\}$ are most important: $\{A_{11}\}$ would imply that buf holds an input read from channel $\bot$ (justifying why copying its contents to the $\bot$ variable $\bot buf$ is secure), and $\{A_{12}\}$ would imply



(a) Data flows. Dotted lines denote declassification.

```
{A0} ⊤ ! ⊤buf
```

(b) Outputting $\top$ data.

```
{A1} acquire(ℓ);
{A2} if inmode = 0
   {A3} buf ← ⊥
else
   {A4} buf ← ⊤
endif;
{A5} valid := 1;
{A6} release(ℓ)
```

(c) Reading data into a shared buffer.

```
{A7} ⊥ ! ⊥buf
```

(d) Outputting $\perp$ data.

```
{A8} acquire(ℓ);
{A9} if valid = 1
  {A10} if outmode = 0
     {A11} ⊥buf := buf
  else
     {A12} ⊤buf := buf;
     {A13} d :≘ CK (⊤buf);
     {A14} if d = 0
        {A15} ⊥buf :≘ ⊤buf
     endif
  endif
endif ;
{A16} release(ℓ)
```

(e) Copying and declassifying data.

Figure 1: Co-operative Use of a Shared Buffer. Green $\{A_i\}$ are functional correctness annotations (whose contents we omit).

likewise for channel $\top$. $\{A_{13}\}$ would imply that $\top buf$ holds $\top$ data and $\{A_{14}\}$ that d holds the result of the signature check. Finally, $\{A_{15}\}$ implies that the signature check passed, justifying why the declassifying assignment to $\perp buf$ is secure.

The other annotations encode functional correctness information needed to justify the validity of the aforementioned annotations. For instance, annotation $\{A_2\}$ in Figure 1c implies that the thread holds the lock ℓ; $\{A_3\}$ that *inmode* is zero, while $\{A_4\}$ the opposite. Annotation $\{A_5\}$ on the other hand tracks information about the contents of *buf*, namely if *inmode* is zero then *buf* holds the last input read from channel $\perp$, and it holds the last input read from channel $\top$ otherwise[1].

Thus the annotations $\{A_i\}$ afford highly precise reasoning about the security of each thread, while decoupling the functional correctness reasoning.

The idea of using annotations $\{A_i\}$ we repurpose from the Owicki-Gries proof technique [50] for concurrent programs. Indeed, there exist a range of standard techniques for inferring and proving the soundness of such annotations (i.e. for carrying out the functional correctness reasoning), from the past 40 years of research on concurrent program verification. VERONICA integrates multiple such techniques in the Isabelle/HOL theorem prover, each of which has been proved sound from first principles, thereby ensuring the correctness of its foundations. As we explain later, external program verifiers may also be used to verify functional correctness, giving up VERONICA's foundational guarantees in exchange for ease of verification.

Given a correctly annotated program, VERONICA then uses the information encoded in the annotations to prove expressive security policies, as we outline in the next section.

2.2. Compositional Enforcement

How can we prove that the concurrent program of Figure 1 doesn't violate information flow security, i.e. that no $\top$ data is leaked, unless it has been declassified in accordance with the delimited release policy?

Doing so in general benefits from having a compositional reasoning method, namely one that reasons over each of the program's threads separately to deduce that the concurrent execution of those threads is secure.

Compositional methods for proving information flow properties of concurrent programs have been studied for decades [20], [21]. Initial methods required one to prove that each thread was secure ignorant of the behaviour of other threads [20], [21], [24]. Such reasoning is sound but necessarily imprecise: for instance when reasoning about the top-middle thread (Figure 1c) we wouldn't be allowed to assume that the top-right thread (Figure 1e) adheres to the locking protocol that protects *buf*.

Following Mantel et al. [30], more modern compositional methods have adopted ideas from rely-guarantee reasoning [35] and concurrent separation logic [51], to allow more precise reasoning about each thread under assumptions it makes about the behaviour of others (e.g. correct locking discipline) [10], [19], [28]. However, the precision of these methods comes at the price of expressiveness: specifically, their inability to reason about declassification. By *decoupling* functional correctness reasoning, VERONICA achieves both precision and expressiveness.

The VERONICA logic—VERONICA's compositional IFC proof method—has judgements of the form $lvl_{\mathsf{A}} \vdash c$, where lvl_{A} is a security level (e.g. $\top$ or $\perp$ in the case of Figure 1) representing level of the attacker and c is a fragment of program text (i.e. a program statement). This judgement holds if the program fragment c doesn't leak information to level lvl_{A} that lvl_{A} should not be allowed to observe. For the code of each thread t, one uses the rules of VERONICA's logic to prove that $lvl_{\mathsf{A}} \vdash c$ holds, where lvl_{A} ranges over all possible security levels. By doing so one establishes that the concurrent program is secure, under the assumption that the concurrent program is functionally correct (i.e. each of its annotations $\{A_i\}$ hold when the concurrent program is executed). As mentioned, functional correctness can be

1. $\{A_5\}$ effectively encodes *buf*'s (state-dependent) sensitivity, and takes the place of dependent security types and labels from prior systems.

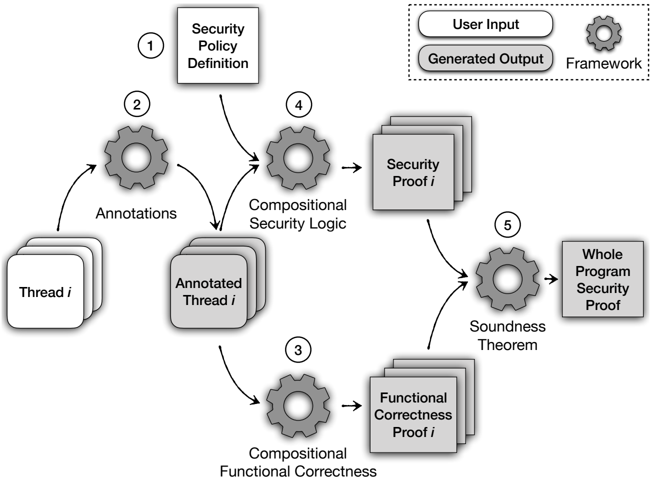


Figure 2: Proving a program secure in VERONICA.

proved using a range of well-established techniques that integrate into VERONICA.

Unlike recent compositional proof methods (c.f. [10], [19], [28], [30]), the judgement of VERONICA has no need to track variable stability information (i.e. which variables won't be modified by other threads), nor any need for a flow-sensitive typing context to track the sensitivity of data in shared program variables, nor does it track constraints on the values of program variables. Instead, this information is provided via the annotations $\{A_i\}$.

For example, the annotation $\{A_{11}\}$ in Figure 1 (Figure 1e, line 4) states that: (1) when $valid$ is 1, if $inmode$ is 0 then buf contains the last input read from channel $\bot$ and otherwise it contains the last $\top$ input; (2) the top-right thread holds the lock ℓ; (3) $inmode$ and $outmode$ agree; and (4) $outmode$ is 0 and $valid$ is 1. Condition (1) implicitly encodes sensitivity information about the data in the shared variable buf; (2) encodes stability information; while (3) and (4) are constraints on shared program variables.

To prove that the assignment on line 4 of Figure 1e is secure, VERONICA requires one to show that the sensitivity of the data contained in buf is at most $\bot$ (the level of $\bot buf$). However one gets to assume that the annotation at this point $\{A_{11}\}$ holds. In this case, the obligation is discharged straightforwardly from the annotation. The same is true for other other parts of this concurrent program. In this way, VERONICA leans on the functional correctness annotations to establish security, and utilises compositional functional correctness to unify reasoning about various security concerns (e.g. declassification, state-dependent sensitivity, etc.).

2.3. Proving a Concurrent Program Secure

Figure 2 depicts the process of proving a concurrent program secure using VERONICA. The circled numbers indicate the main steps and their ordering.

Step ①: Defining the Security Policy

The first step is to define the security policy that is to be enforced. This involves two tasks. The first is to choose an appropriate lattice of security levels [52] and then to assign security levels to shared variables (e.g. in the example of Figure 1, $\bot buf$ and d both have level $\bot$, while $\top buf$ has level $\top$). A variable's security level is given by the (user supplied) function $\mathcal{L}$, which assigns levels to variables. For variable v, $\mathcal{L}(v)$ defines the maximum sensitivity of the data that v is allowed to hold at all times.

In VERONICA not all shared variables need be assigned a security level, meaning that $\mathcal{L}$ is allowed to be a *partial* function. For instance, in the example of Figure 1, buf has no level assigned (i.e. $\mathcal{L}(buf)$ is undefined). The security policy does not restrict the sensitivity of the data that such *unlabelled* variables are allowed to hold. This is useful for shared variables like buf that form the interface between two threads and whose sensitivity is governed by a data-dependent contract [10]. In the example, this allows buf (whenever $valid$ is 1) to hold $\bot$ data when $inmode$ and $outmode$ are both zero, and $\top$ data when $inmode$ and $outmode$ are both nonzero.

The second part of defining the security policy is to specify when and how declassification is allowed to occur. In order to maximise expressiveness, VERONICA supports dynamic, state-dependent declassification policies. Such policies are encoded via the (user supplied) predicate $\mathcal{D}$. For a source security level lvl_{src} and destination security level lvl_{dst}, the program command c is allowed to declassify the lvl_{src}-sensitivity value v to level lvl_{dst} in system state σ precisely when $\mathcal{D}(lvl_{src}, lvl_{dst}, \sigma, v, c)$ holds. Note that the command c is either a declassifying assignment "$\{A_i\}$ $x \mathrel{\widehat{:=}} E$" (in which case lvl_{dst} is the label $\mathcal{L}(x)$ assigned to the labelled variable x) or a declassifying output "$\{A_i\}$ $lvl_{dst} \mathrel{\widehat{!}} E$". In either case, lvl_{src} is the security level of the expression E and v is the result of evaluating E in state σ.

This style of declassification predicate is able to support various declassification policies, including delimited release style policies as in the example of Figure 1. We discuss precisely how delimited release policies are encoded as declassification predicates $\mathcal{D}$ later in Section 3.4. Other declassification policies are encountered in Section 6.

Step ②: Supply Annotations

Having defined the security policy, the second step to proving a concurrent program secure using VERONICA is to supply sufficient functional correctness annotations $\{A_i\}$ for each thread. In the example of Figure 1, while their contents is not shown, these annotations are already present. However in practice, users of VERONICA will start with un-annotated programs for which functional correctness annotations $\{A_i\}$ are then supplied to decorate statements c of each thread, encoding what facts are believed to be true about the state of the concurrent program whenever statement c executes.

Note that, because these annotations will be verified later (in step ③), there is no need to trust the process that generates them. The current Isabelle incarnation of VERONICA includes a proof-of-concept strongest-postcondition style annotation inference algorithm, whose results can then be manually tweaked by the user as necessary. Users are also free to employ external, automatic program analysis tools to infer functional correctness annotations, or to supply annotations manually, without fear of compromising the foundational guarantees of VERONICA.

Step ③: Verifying Functional Correctness

Having obtained the functional correctness annotations $\{A_i\}$, the next step is to *prove* their validity. This means proving that the concurrent program is functionally correct, for which there exist numerous compositional techniques [35], [50].

VERONICA incorporates two standard techniques in the Isabelle/HOL formalisation: the Owicki-Gries method [50] and Rely-Guarantee reasoning [35]. VERONICA's Owicki-Gries implementation is borrowed from the seminal work of Prensa Nieto [53], [54]. Using it to verify (correct) functional correctness annotations requires little effort for experienced Isabelle users, by guiding Isabelle's proof automation tactics. Like VERONICA's Owicki-Gries method, its Rely-Guarantee implementation is for verifying functional correctness annotations only, and ignores security (c.f. [19], [30]). It requires the user to supply *rely* and *guarantee* conditions for each thread. Such conditions can be defined straightforwardly from the locking protocol of a concurrent program and in principle could be inferred; however, we leave that inference for future work.

If one wishes to forego the foundational assurance of Isabelle/HOL, one can also employ external verification tools to prove annotation validity. By doing so one may elide *non-essential* annotations, which are those on statements other than output statements, declassifications, assignments to labelled variables, and branches on unlabelled data. Our formalisation includes a proof-of-concept of this approach, in which the concurrent C program verifier VCC [55] is employed on a C translation of the example in Figure 1 to prove the functional correctness of its essential annotations, sufficient to guarantee its security.

Step ④: Verifying Security

With functional correctness proved, the user is then free to use the functional correctness annotations to compositionally prove the security of the concurrent program. To do this, the user applies the rules of the VERONICA logic to each of the program's threads. VERONICA exploits the functional correctness assertions to provide a simple logic resembling a flow-insensitive type system. Each statement is verified independently of its context. The logic is compositional and allows each thread to be verified in isolation. Rules for output statements require that the functional correctness annotations imply that the expression always evaluates to the same result given states that are not distinguishable to the attacker. Similarly, the rules for declassification statements require that the annotations are sufficient to imply that the declassification predicate holds. We defer a full presentation of the logic to Section 5.

Step ⑤: Whole Program Security Proof

With both functional correctness and security proved of each thread, the soundness theorem of the VERONICA logic can then be applied to derive a theorem stating that the whole concurrent program is secure. This theorem is stated formally in Section 5.3. However, intuitively it says that the whole concurrent program is secure if, for each thread t, t's functional correctness annotations are all valid (i.e. each holds whenever the corresponding statement of t is executed in the concurrent program)—step ③—and t is judged secure by the rules of the VERONICA logic—step ④.

3. Security Definition

VERONICA proves an information flow security property designed to capture a range of different security policies. To maximise generality, the security property is phrased in a knowledge-based (or *epistemic*) style, which as others have argued [13], [14], [56], [57] is preferable to traditional two-run formulations. Before introducing the security property and motivating the threat model that it formally encodes, we first explain the semantic model of concurrent program execution in which the property is defined.

Along the way, we highlight the assumptions encoded in that semantic model and in the formal security property. Following Murray and van Oorschot [58], we distinguish *adversary expectations*, which are assumptions about the attacker (e.g. their observational powers); from *domain hypotheses*, which are assumptions about the environment (e.g. the scheduler) in which the concurrent program executes.

3.1. Semantic Model

Concurrent programs comprise a finite collection of n threads, each of which is identified by a natural number: $0, \ldots, n-1$. Threads synchronise by acquiring and releasing locks and communicate by modifying shared memory. Additionally, threads may communicate with the environment outside the concurrent program by inputting and outputting values from/to IO *channels*. Without loss of generality, there is one channel for each security level (drawn from the user-supplied lattice of security levels).

Global States σ

Formally, the *global states* σ of the concurrent program are tuples $(env_\sigma, mem_\sigma, locks_\sigma, tr_\sigma)$. The global state contains all resources that are shared between threads. We consider each in turn.

Channels and the Environment env_σ

env_σ captures the state of the external environment (i.e. the IO channels). For a security level lvl, $env_\sigma(lvl)$ is the (infinite) stream of values yet to be consumed from the channel lvl in state σ.

Domain Hypothesis In this model of channels, reading from a channel never blocks and always returns the next value to be consumed from the infinite stream. This effectively assumes that all channel inputs are faithfully buffered and never dropped by the environment. Blocking can be simulated by repeatedly polling a channel.

Shared Memory mem_σ

mem_σ is simply a total mapping form variable names (excluding locks) to their corresponding values: $mem_\sigma(v)$ denotes the value of variable v in state σ.

Locks $locks_\sigma$

$locks_\sigma$ captures the lock state and is a partial function from lock names to thread ids (natural numbers in the range $0 \ldots n-1$): for a lock ℓ, $locks_\sigma(\ell)$ is defined iff lock ℓ is currently held in state σ, in which case its value is the id of the thread that holds the lock.

Events e **and Traces** tr_σ

For the sake of expressiveness, we store in the global state σ the entire history of events tr_σ that has been performed by the concurrent program up to this point. Each such history is called a *trace*, and is simply a finite list of events e. Events e comprise: *input* events $\mathbf{in}\langle lvl, v\rangle$ which record that value v was input from the channel lvl; *output* events $\mathbf{out}\langle lvl, v, E\rangle$ which record that value v, the result of evaluating expression E, was output on channel lvl; and *declassification* events $\mathbf{d}\langle lvl, v, E\rangle$ which record that the value v, the result of evaluating expression E, was declassified to level lvl. Expression E is included to help specify the security property (see e.g. Definition 3.2.7).

Ordinary (non-declassifying) output and input commands produce output and input events respectively. Declassifying assignments and declassifying outputs produce declassification events. As with much prior work on declassification properties [59], declassification actions produce distinguished declassification events that make them directly visible to the security property.

The Schedule $sched$

The *schedule* $sched$ is an infinite list (stream) of thread ids i. Scheduling occurs by removing the first item i from the stream and then executing the thread i for one step of execution. (Longer execution slices can of course be simulated by repeating i in the schedule.) This process is repeated *ad infinitum*. If thread i is stuck (e.g. because it is waiting on a lock or has terminated) then the system idles (i.e. does nothing) for an execution step, to mitigate scheduling leaks (e.g. as implemented in seL4 [60]).

Domain Hypothesis VERONICA assumes deterministic, sequentially-consistent, instruction-based scheduling [61] (IBS) of threads against a fixed, public schedule.

Global Configurations and Concurrent Execution $\cdot \to \cdot$

A *global configuration* combines the shared global state σ with the schedule $sched$ and the local state ls_i (the thread id and code) of each of the n threads. Thus a global configuration is a tuple: $(ls_0, \ldots, ls_{n-1}, \sigma, sched)$.

Concurrent execution, and the aforementioned scheduling model, is formally defined by the rules of Figure 7 (relegated to the appendix for brevity). These rules define a single-step relation $\cdot \to \cdot$ on global configurations. Zero- and multi-step execution is captured in the usual way by the reflexive, transitive closure of this relation, written $\cdot \to^* \cdot$.

3.2. System Security Property and Threat Model

We now define VERONICA's formal security property, formalising the threat model and adversary expectations.

Attacker Observations

Adversary Expectation: Our security property considers a passive attacker observing the execution of the concurrent program. We assume that the attacker is able to observe outputs on certain channels and associated declassification events. Specifically, the attacker is associated with a security level lvl_A. Outputs on all channels $lvl \leq lvl_\mathsf{A}$ the attacker is assumed to be able to observe. Likewise all declassifications to levels $lvl \leq lvl_\mathsf{A}$.

Adversary Expectation: The attacker has no other means to interact with the concurrent program, e.g. by modifying its code. We additionally assume that the attacker does not have access to timing information.

The attacker's observational powers are formalised by defining a series of indistinguishability relations as follows.

Definition 3.2.1 (Event Visibility)**.** *We say that an input event* $\mathbf{in}\langle lvl, v\rangle$ *(respectively output event* $\mathbf{out}\langle lvl, v, E\rangle$ *or declassification event* $\mathbf{d}\langle lvl, v, E\rangle$*) is* visible *to the attacker at level* lvl_A *iff* $lvl \leq lvl_\mathsf{A}$*. Letting* e *be the event, in this case we write* $visible_{lvl_\mathsf{A}}(e)$*.*

Trace indistinguishability is then defined straightforwardly, noting that we write $tr \upharpoonright P$ to denote filtering from trace tr all events that do not satisfy the predicate P.

Definition 3.2.2 (Trace Indistinguishability)**.** *We say that two traces* tr *and* tr' *are indistinguishable to the attacker at level* lvl_A*, when* $tr \upharpoonright visible_{lvl_\mathsf{A}} = tr' \upharpoonright visible_{lvl_\mathsf{A}}$*.*

In this case, we write $tr \overset{lvl_\mathsf{A}}{\approx} tr'$*.*

Attacker Knowledge of Initial Global State

Besides defining what the attacker is assumed to observe (via the indistinguishability relation on traces), we also need to define what knowledge the attacker is assumed to have about the initial global state σ_init of the system.

Adversary Expectation: The attacker is assumed to know the contents that will be input from channels at levels $lvl \leq lvl_\mathsf{A}$ and the initial values of all labelled variables v for which $\mathcal{L}(v) \leq lvl_\mathsf{A}$.

This assumption is captured via an indistinguishability relation on global states σ. This relation is defined by first defining indistinguishability relations on each of σ's components.

Definition 3.2.3 (Environment Indistinguishability)**.** *We say that two environments env and env' are indistinguishable to the attacker at level lvl_{A} when all channels visible to the attacker have identical streams, i.e. iff*

$$\forall lvl \leq lvl_{\mathsf{A}}.\ env(lvl) = env'(lvl).$$

In this case we write $env \overset{lvl_{\mathsf{A}}}{\approx} env'$.

Definition 3.2.4 (Memory Indistinguishability)**.** *We say that two memories mem and mem' are indistinguishable to the attacker at level lvl_{A} when they agree on the values of all labelled variables v visible to the attacker, i.e. iff*

$$\forall v.\ \mathcal{L}(v) \leq lvl_{\mathsf{A}} \implies mem(v) = mem'(v),$$

where $\mathcal{L}(v) \leq lvl_{\mathsf{A}}$ implies $\mathcal{L}(v)$ is defined.
In this case, we write $mem \overset{lvl_{\mathsf{A}}}{\approx} mem'$.

We can now define when two (initial) global states are indistinguishable to the attacker.

Definition 3.2.5 (Global State Indistinguishability)**.** *We say that two global states σ and σ' are indistinguishable to the attacker at level lvl_{A} iff*

$$env_\sigma \overset{lvl_{\mathsf{A}}}{\approx} env_{\sigma'} \wedge mem_\sigma \overset{lvl_{\mathsf{A}}}{\approx} mem_{\sigma'} \wedge$$
$$locks_\sigma = locks_{\sigma'} \wedge tr_\sigma \overset{lvl_{\mathsf{A}}}{\approx} tr_{\sigma'}$$

In this case we write $\sigma \overset{lvl_{\mathsf{A}}}{\approx} \sigma'$.

Domain Hypothesis Under this definition, the attacker knows the entire initial lock state. Thus we assume that the initial lock state encodes no secret information.

Attacker Knowledge from Observations

Given the attacker's knowledge about the initial state σ_{init} and some observation arising from some trace tr being performed, we assume that the attacker will then attempt to refine their knowledge about σ_{init}.

Adversary Expectation: The attacker is assumed to know the schedule $sched$ and the initial local state ls_i (i.e. the code and thread id i) of each thread.

Given that information, of all the possible initial states from which σ_{init} might have been drawn, perhaps only a subset can give rise to the observation of tr. We assume the attacker will perform this kind of knowledge inference, which we formalise following the epistemic style [56].

To define the attacker's knowledge, we define the attacker's uncertainty about the initial state σ_{init} (i.e. the attacker's belief about the set of all initial states from which σ_{init} might have been drawn) given the initial schedule $sched$ and local thread states $ls_0, \ldots, ls_{n-1}$, and the trace tr that the attacker has observed. Writing simply ls to abbreviate the list $ls_0, \ldots, ls_{n-1}$, we denote this $uncertainty_{lvl_{\mathsf{A}}}(ls, \sigma_{\mathsf{init}}, sched, tr)$ and define it as follows.

Definition 3.2.6 (Attacker Uncertainty)**.** *A global state σ belongs to the set $uncertainty_{lvl_{\mathsf{A}}}(ls, \sigma_{\mathsf{init}}, sched, tr)$ iff it and σ_{init} are indistinguishable, given the attacker's knowledge about the initial state, and if σ can give rise to a trace $tr_{\sigma'}$ that is indistinguishable from tr. Formally, $uncertainty_{lvl_{\mathsf{A}}}(ls, \sigma_{\mathsf{init}}, sched, tr)$ is the set of σ where*

$$\begin{aligned}&\sigma \overset{lvl_{\mathsf{A}}}{\approx} \sigma_{\mathsf{init}} \wedge \\ &\exists ls'\ \sigma'\ sched'.\ (ls, \sigma, sched) \rightarrow^* (ls', \sigma', sched') \wedge \\ &\qquad\qquad tr \overset{lvl_{\mathsf{A}}}{\approx} tr_{\sigma'}\end{aligned}$$

The Security Property

Finally, we can define the security property. This requires roughly that the attacker's uncertainty can decrease (i.e. they can refine their knowledge) only when declassification events occur, and that all such events must respect the declassification policy encoded by $\mathcal{D}$. In other words, the guarantee provided by VERONICA under the threat model formalised herein is that:

Security Guarantee: The attacker is never able to learn any new information above what they knew initially, *except* from declassification events but those must *always* respect the user-supplied declassification policy.

This guarantee is formalised by defining a *gradual release*-style security property [56]. We first define when the occurrence of an event e is secure.

Definition 3.2.7 (Event Occurrence Security)**.** *Consider an execution beginning in some initial configuration $(ls, \sigma, sched)$ that has executed to the intermediate configuration $(ls', \sigma', sched')$ from which the event e occurs. This occurrence is secure against the attacker at level lvl_{A}, written $esec_{lvl_{\mathsf{A}}}((ls, \sigma, sched), (ls', \sigma', sched'), e)$, iff*

- *When e is a declassification event $\mathbf{d}\langle lvl_{dst}, v, E\rangle$ visible to the attacker (i.e. $lvl_{dst} \leq lvl_{\mathsf{A}}$), then $\mathcal{D}(\mathcal{L}(E), lvl_{dst}, \sigma', v, c)$ must hold, where c is the current program command whose execution produced e (i.e. the head program command of the currently executing thread in $(ls', \sigma', sched')$). Here, $\mathcal{L}(E)$ is defined when $\mathcal{L}(v)$ is defined for all variables v mentioned in E and in that case is the least upper bound of all such $\mathcal{L}(v)$, and $\mathcal{D}(\mathcal{L}(E), lvl_{dst}, \sigma', v, c)$ is false when $\mathcal{L}(E)$ is not defined.*
- *Otherwise, if e is not a declassification event $\mathbf{d}\langle lvl_{dst}, v, E\rangle$ that is visible to the attacker, then the attacker's uncertainty cannot decrease by observing it, i.e. we require that*

$$\begin{aligned}&uncertainty_{lvl_{\mathsf{A}}}(ls, \sigma, sched, tr_{\sigma'}) \subseteq \\ &\quad uncertainty_{lvl_{\mathsf{A}}}(ls, \sigma, sched, tr_{\sigma'} \cdot e)\end{aligned}$$

Definition 3.2.8 (System Security)**.** *The concurrent program with initial local thread states $ls = (ls_0, \ldots, ls_{n-1})$ is secure against an attacker at level lvl_{A}, written $syssec_{lvl_{\mathsf{A}}}(ls)$, iff, under all schedules $sched$, event occurrence security* *always holds during its execution from any initial starting state σ. Formally, we require that*

$$\begin{aligned}&\forall sched\ \sigma\ ls'\ \sigma'\ sched'\ ls''\ \sigma'' sched''\ e.\\ &\quad (ls, \sigma, sched) \to^* (ls', \sigma', sched') \wedge\\ &\quad (ls', \sigma', sched') \to (ls'', \sigma'', sched'') \wedge\\ &\quad tr_{\sigma''} = tr_{\sigma'} \cdot e \implies\\ &\qquad esec_{lvl_{\mathsf{A}}}((ls, \sigma, sched), (ls', \sigma', sched'), e)\end{aligned}$$

3.3. Discussion

As with other gradual release-style properties, ours does not directly constrain *what* information the attacker might learn when a declassification event occurs, but merely that those are the only events that can increase the attacker's knowledge. This means that, of the four semantic principles of declassification identified by Sabelfeld and Sands [62], our definition satisfies all but *non-occlusion*: "the presence of declassifications cannot mask other covert information leaks" [62]. Consider the following single-threaded program.

$\{A_{17}\}$ **if** $birthYear > 2000$
  $\{A_{18}\}$ $\bot \hat{!}\, birthDay$
else
  $\{A_{19}\}$ $\bot \hat{!}\, birthMonth$

Suppose the intent is to permit the unconditional release of a person's day and month of birth, but not their birth year. A naive encoding in the declassification policy $\mathcal{D}$ that checks whether the value being declassified is indeed either the value of $birthDay$ or $birthMonth$ would judge the above program as secure, when in fact it also leaks information about the $birthYear$.

Note also, since declassification events are directly visible to our security property, that programs that incorrectly declassify information but then never output it on a public channel can be judged by our security condition as insecure.

Finally, and crucially, note that our security condition allows for both *extensional* declassification policies, i.e. those that refer only to inputs and outputs of the program, as well as *intensional* policies that also refer to the program state. Section 6 demonstrates both kinds of policies. We now consider one class of extensional policies: delimited release.

3.4. Encoding Delimited Release Policies

The occlusion example demonstrates that programs that branch on secrets that are not allowed to be released and then perform declassifications under that secret context are likely to leak more information than that contained in the declassification events themselves, via implicit flows.

However, in the absence of such branching, our security condition can in fact place bounds on what information is released. Specifically, we show that it can soundly encode delimited release [45] policies as declassification predicates $\mathcal{D}$ for programs that do not branch on secrets that are not allowed to be declassified to the attacker.

We define an extensional delimited release-style security condition and show how to instantiate the declassification predicates $\mathcal{D}$ so that when system security (Definition 3.2.8) holds, then so does the delimited release condition.

3.4.1. Formalising Delimited Release

Delimited release [45] weakens traditional noninterference [34] by permitting certain secret information to be released to the attacker. Which secret information is allowed to be released is defined in terms of a set of *escape hatches*: expressions that denote values allowed to be released.

Delimited release then strengthens the indistinguishability relation on the initial state to require that any two states related under this relation also agree on the values of the escape hatch expressions. One way to understand delimited release as a weakening of noninterference is to observe that, in changing the relation in this way, it is effectively encoding the assumption that the attacker might *already know* the secret information denoted by the escape hatch expressions.

To keep our formulation brief, we assume that the initial memory contains no secrets. Thus all secrets are contained only in the input streams (channels). Then escape hatches denote values that are allowed to be released as functions on lists vs of inputs (to be) consumed from a channel.

A *delimited release* policy $\mathcal{E}$ is a function that given source and destination security levels lvl_{src} and lvl_{dst} returns a set of escape hatches denoting the information that is allowed to be declassified from level lvl_{src} to level lvl_{dst}.

For example, to specify that the program is always allowed to declassify to $\bot$ the average of the last five inputs read from the $\top$ channel, one could define $\mathcal{E}(\top, \bot) = \{\lambda vs.\ \mathbf{if}\ len(vs) \geq 5\ \mathbf{then}\ avg(take(5, rev(vs)))\ \mathbf{else}\ 0\}$, where $avg(xs)$ calculates the average of a list of values xs, $take(n, xs)$ returns a new list containing the first n values from the list xs, and $rev(xs)$ is the list reversal function.

To define delimited release, we need to define when two initial states σ agree under the escape hatches $\mathcal{E}$. Since escape hatches apply only to the streams contained in the environment env_σ, we define when two such environments agree under $\mathcal{E}$. As earlier, this agreement is defined relative to an attacker observing at level lvl_{A}, and requires that all escape hatches that yield values that the attacker is allowed to observe always evaluate identically under both environments.

Definition 3.4.1 (Environment Agreement under $\mathcal{E}$)**.** *Two environments env and env' agree under the delimited release policy $\mathcal{E}$ for an attacker at level lvl_{A}, written $env \overset{lvl_{\mathsf{A}}, \mathcal{E}}{\approx} env'$, iff, for all levels lvl_{src} and all levels $lvl_{dst} \leq lvl_{\mathsf{A}}$, and escape hatches $h \in \mathcal{E}(lvl_{src}, lvl_{dst})$, h applied to any finite prefix of $env(lvl_{src})$ yields the same value as when applied to an equal length prefix of $env'(lvl_{src})$.*

We then define when two initial states σ and σ' agree for a delimited release policy $\mathcal{E}$. The following definition is a slight simplification of the one in our Isabelle formalisation (see Definition A.1 in the appendix), which is more general because it considers arbitrary pairs of states in which some trace of events might have already been performed.

Definition 3.4.2 (State Agreement under $\mathcal{E}$)**.** *States σ and σ' agree under the delimited release policy $\mathcal{E}$ for an attacker at level lvl_{A}, written $\sigma \overset{lvl_{\mathsf{A}}, \mathcal{E}}{\approx} \sigma'$, iff (1) $\sigma \overset{lvl_{\mathsf{A}}}{\approx} \sigma'$, (2) their* *memories agree on all variables, and (3)* $env_\sigma \overset{lvl_{\mathsf{A}},\mathcal{E}}{\approx} env_{\sigma'}$.

Here, condition (2) encodes the simplifying assumption that the initial memories contain no secrets.

Delimited release is then defined extensionally in the style of traditional two-run noninterference property.

Definition 3.4.3 (Delimited Release)**.** *The concurrent program with initial local thread states* $ls = (ls_0, \ldots, ls_{n-1})$ *satisfies* delimited release *against an attacker at level* lvl_{A}*, written* $drsec_{lvl_{\mathsf{A}}}(ls)$*, iff,*

$$\forall sched\ \sigma\ \sigma'\ y.\ \sigma \overset{lvl_{\mathsf{A}},\mathcal{E}}{\approx} \sigma' \wedge (ls, \sigma, sched) \to^* y \implies (\exists y'.\ (ls, \sigma', sched) \to^* y' \wedge tr_y \overset{lvl_{\mathsf{A}}}{\approx} tr_{y'})$$

where for a global configuration $y = (ls_y, \sigma_y, sched_y)$ *we write* tr_y *to abbreviate* tr_{σ_y}*, the trace executed so far.*

3.4.2. Encoding Delimited Release in $\mathcal{D}$

We now encode delimited release policies $\mathcal{E}$ via VERONICA's declassification predicates $\mathcal{D}(lvl_{src}, lvl_{dst}, \sigma, v, c)$ which, recall, judge whether command c declassifying value v from level lvl_{src} to level lvl_{dst} in state σ is permitted. Recall that c is either a declassifying assignment "$\{A_i\}$ $x \mathrel{\widehat{:=}} E$" (in which case lvl_{dst} is the label $\mathcal{L}(x)$ assigned to the labelled variable x) or a declassifying output "$\{A_i\}$ $lvl_{dst} \mathrel{\widehat{!}} E$". In either case, lvl_{src} is the security level of the expression E and v is the result of evaluating E in state σ.

To encode delimited release, we need to have $\mathcal{D}(lvl_{src}, lvl_{dst}, \sigma, v, c)$ decide whether there is an escape hatch $h \in \mathcal{E}(lvl_{src}, lvl_{dst})$ that permits the declassification. Consider some $h \in \mathcal{E}(lvl_{src}, lvl_{dst})$. What does it mean for h to permit the declassification? Perhaps surprisingly, it is *not enough* to check whether h evaluates to the value v being declassified in σ. Suppose h permits declassifying the average of the last five inputs from channel $\top$ and suppose in σ that this average is 42. An insecure program might declassify some *other* secret whose value just happens to be 42 in σ, but that declassification would be unlikely to satisfy delimited release if the two secrets are independent.

Instead, to soundly encode delimited release, one needs to check whether the expression E being declassified is equal to the escape hatch *in general*.

To do this, we have $\mathcal{D}$ check that in all states in which this declassification c might be performed, the escape hatch h evaluates to the value of E in that state. We can overapproximate the set of all states in which c might execute by using its annotation $\{A_i\}$: all such states must satisfy the annotation assuming the program is functionally correct (which VERONICA will prove). Thus we have $\mathcal{D}$ check that in all such states that satisfy the annotation, the escape hatch h evaluates to the expression E.

Definition 3.4.4 (Delimited Release Encoding)**.** *The encoding of policy* $\mathcal{E}$ *we denote* $\mathcal{D}_\mathcal{E}$*.* $\mathcal{D}_\mathcal{E}(lvl_{src}, lvl_{dst}, \sigma, v, c)$ *holds always when* c *is not a declassification command. Otherwise, let* A *be* c*'s annotation and* E *be the expression that* c *declassifies. Then* $\mathcal{D}_\mathcal{E}(lvl_{src}, lvl_{dst}, \sigma, v, c)$ *holds iff there exists some* $h \in \mathcal{E}(lvl_{src}, lvl_{dst})$ *such that for all states* σ' *that satisfy the annotation* A*,* E *evaluates in* σ' *to the same value that* h *evaluates to when applied to the* lvl_{src} *inputs consumed so far in* σ'*.*

Recall this encoding is sound only for programs that do not branch on secrets that the policy $\mathcal{E}$ forbids from releasing. We define this condition semantically as a two-run property, relegating it to Definition A.2 in the appendix since its meaning is intuitively clear. We say that a program satisfying this condition is *free of* $\mathcal{E}$*-secret branching*.

The example of Section 3.3 that leaks $birthYear$ via occlusion is not free of $\mathcal{E}$-secret branching. On the other hand, the program in Figure 1 is free of $\mathcal{E}$-secret branching for the following $\mathcal{E}$ that defines its delimited release policy, since the only $\top$-value ever branched on (in Figure 1e, line 8) is the result of the signature check CK.

Definition 3.4.5 (Delimited Release policy for Figure 1)**.** *Allow to be declassified to* $\bot$ *the results of the signature check* CK *always, plus any* $\top$*-input* v *when* $CK(v) = 0$*.*

$$\begin{aligned}&\mathcal{E}(\top, \bot) = \\ &\quad \{\lambda vs.\ \textbf{if}\ len(vs) \neq 0\ \textbf{then}\ CK(last(vs))\ \textbf{else}\ 0\} \cup \\ &\quad \{\lambda vs.\ \textbf{if}\ len(vs) \neq 0 \wedge CK(last(vs)) = 0\ \textbf{then} \\ &\quad\ \ last(vs)\ \textbf{else}\ 0\}\end{aligned}$$

Indeed, VERONICA can be used to prove that Figure 1 satisfies this delimited release policy by showing that it satisfies VERONICA's system security (Definition 3.2.8), under the following theorem that formally justifies why VERONICA can encode delimited release policies.

Theorem 3.4.1 (Delimited Release Embedding)**.** *Let* lvl_{A} *be an arbitrary security level and* ls *be the initial local thread states (i.e. thread ids and the code) of a concurrent program that (1) satisfies* $syssec_{lvl_{\mathsf{A}}}(ls)$ *with* $\mathcal{D}$ *defined according to Definition 3.4.4, (2) is free of* $\mathcal{E}$*-secret branching, and (3) satisfies all of its functional correctness annotations. Then, the program is delimited release secure, i.e.* $drsec_{lvl_{\mathsf{A}}}(ls)$*.*

Thus VERONICA can soundly encode purely extensional security properties like Definition 3.4.3. The extensional form of the policy for the Figure 1 example is straightforward and relegated to the extended version of this paper [44].

4. Annotated Programs in VERONICA

VERONICA reasons about the security of concurrent programs, each of whose threads is programmed in the language whose grammar is given in Figure 3.

Most of these commands are straightforward and appear in Figure 1. Loops "$\{A\}$ **while** E **inv** $\{I\}$ **do** c" carry a second *invariant* annotation (here "$\{I\}$") that specifies the loop invariant, which is key for proving their functional correctness [63]. The "**stop**" command halts the execution of the thread, and is an internal form used only to define the semantics of the language. The *no-op* command "$\{A\}$ **nop**" is syntactic sugar for: "$\{A\}$ $x := x$", while "$\{A\}$ **if** E c **endif**" is sugar for "$\{A\}$ **if** E c **else** $\{A\}$ **nop endif**".

The semantics for this sequential language is given in Figure 8, and is relegated to the appendix since it is straightforward. This semantics is defined as a small step

c ::= $\{A\}\ x := E$	(assignment)
$\mid \{A\}\ x \mathrel{\widehat{:=}} E$	(declassifying assignment)
$\mid \{A\}\ lvl\,!\,E$	(output to channel lvl)
$\mid \{A\}\ lvl\,\widehat{!}\,E$	(declassifying output)
$\mid \{A\}\ x \leftarrow lvl$	(input from channel lvl)
$\mid \{A\}$ **if** E c **else** c **endif**	(conditional)
$\mid \{A\}$ **while** E **inv** $\{A\}$ **do** c	(loop with invariant)
$\mid \{A\}$ **acquire**(ℓ)	(lock acquisition)
$\mid \{A\}$ **release**(ℓ)	(lock release)
$\mid c;\ c$	(sequencing)
$\mid$ **stop**	(terminated thread)

Figure 3: Syntax of VERONICA threads.

relation on *local configurations* (ls_i, σ) where $ls_i = (i, c)$ is the local state (thread id i and code c) for a thread and $\sigma = (env_\sigma, mem_\sigma, locks_\sigma, tr_\sigma)$ is the global state shared with all other threads. $\lfloor E \rfloor_{mem_\sigma}$ is atomic evaluation of expression E in memory mem_σ. Notice that the semantics doesn't make use of the annotations $\{A\}$: they are merely decorations used to decouple functional correctness.

5. The VERONICA Logic

The VERONICA logic defines a compositional method to prove when a concurrent program satisfies system security (Definition 3.2.8), VERONICA's security condition. Specifically, it defines a set of rules for reasoning over the program text of each thread of the concurrent program. A soundness theorem (Theorem 5.3.1) guarantees that programs that are functionally correct and whose threads are proved secure using the VERONICA logic satisfy system security.

The rules of the VERONICA logic appear in Figure 4. They define a judgement resembling that for a flow-insensitive security type system that has the form: $lvl_{\mathsf{A}} \vdash c$ where lvl_{A} is the attacker level and c is an annotated thread command (see Figure 3).

5.1. Precise Reasoning with Annotations

The rules for VERONICA explicitly make use of the annotations $\{A\}$ on program commands to achieve highly precise reasoning, while still presenting a simple logic to the user. This is evident in the simplicity of many of the rules of Figure 4. To understand how annotations are used to achieve precise reasoning, consider the rule OUTTY for outputting on channel lvl. When this output is visible to the attacker ($lvl \le lvl_{\mathsf{A}}$), this rule uses the annotation A to reason about the sensitivity of the data contained in the expression E at this point in the program, specifically to check that this sensitivity is no higher than the attacker level lvl_{A}. This is captured by the predicate $sensitivity(A, E, lvl_A)$.

For a security level lvl, annotation A and expression E, $sensitivity(A, E, lvl)$ holds when, under A, the sensitivity of the data contained in E is not greater than lvl. This is *not* a policy statement about $\mathcal{L}(E)$ but, rather, uses A to over-approximates E's sensitivity at this point in the program.

$$sensitivity(A, E, lvl) \equiv \forall \sigma\ \sigma'.\ \sigma \vDash A \wedge \sigma' \vDash A \wedge \sigma \overset{lvl}{\approx} \sigma' \implies \lfloor E \rfloor_{mem_\sigma} = \lfloor E \rfloor_{mem_{\sigma'}}$$

Similarly, the declassification rules DASGTY and DOUTTY use the annotation A to reason precisely about whether $\mathcal{D}$ holds at this point during the program.

5.2. Secret-Dependent Branching

The rule WHILETY for loops does not allow the loop guard to depend on a secret. In contast, the rule IFTY for reasoning about conditionals "$\{A\}$**if** E c_1 **else** c_2 **endif**"does. Its final premise appplies when the sensitivity of the condition E exceeds that which can be observed by the attacker lvl_{A}, i.e. when the if-condition has branched on a secret that should not be revealed to the attacker. The premise requires proving that the two branches c_1 and c_2 are lvl_{A}*-bisimilar*, i.e. that the attacker cannot distinguish the execution of c_1 from the execution of c_2. The formal definition of bisimilarity (Definition A.5) appears in the appendix.

VERONICA includes a set of proof rules to determine whether two commands are lvl_{A}-bisimiar. These rules have been proved sound but, due to lack of space, we refer the reader to our Isabelle formalisation for the full details. Briefly, these rules check that both commands (1) perform the same number of execution steps, (2) modify no labelled variables x for which $\mathcal{L}(x) \le lvl_{\mathsf{A}}$, (3) never input from or output to channels $lvl \le lvl_{\mathsf{A}}$, and (4) perform no declassifications. Thus the lvl_{A}-attacker cannot tell which command was executed, including via scheduling effects.

One is of course free to implement other analyses to determine bisimilarity. Hence, VERONICA provides a modular interface for reasoning about secret-dependent branching.

5.3. Soundness

Recall that the soundness theorem requires the concurrent program (with initial thread states) $ls = (ls_0, \ldots, ls_{n-1})$ to satisfy all of its functional correctness annotations. When this is the case we write $\vDash ls$.

Theorem 5.3.1 (Soundness). *Let* $ls = ((0, c_0), \ldots, (n-1, c_{n-1}))$ *be the initial local thread states of a concurrent program. If* $\vDash ls$ *holds and* $lvl_{\mathsf{A}} \vdash c_i$ *holds for all* $0 \le i < n$*, then the program satisfies system security, i.e.* $syssec_{lvl_{\mathsf{A}}}(ls)$*.*

In practice one applies the VERONICA logic for an arbitrary attacker security level lvl_{A}, meaning that system security will hold for attackers at all security levels.

The condition $\vDash ls$ can be discharged using any of the techniques implemented in VERONICA (see Figure 2.3; Step ③), or via any sound correctness verification method.

6. Further Examples

6.1. The Example of Figure 1

Recall that the concurrent program of Figure 1 implements an extensional delimited release style policy $\mathcal{E}$ defined in Definition 3.4.5 .

We add a fifth thread, which toggles $inmode$ and $outmode$ while ensuring they agree, and sets $valid$ to zero.

$\{A_{20}\}$ **acquire**(ℓ);
$\{A_{21}\}$ $valid := 0$;

$$\frac{\mathit{lvl}_\mathsf{A} \vdash c_1 \qquad \mathit{lvl}_\mathsf{A} \vdash c_2}{\mathit{lvl}_\mathsf{A} \vdash c_1;\ c_2}\textsc{SeqTy} \qquad \frac{\mathcal{L}(x) \text{ is undefined}}{\mathit{lvl}_\mathsf{A} \vdash \{A\} x := E}\textsc{UAsgTy} \qquad \frac{\mathit{sensitivity}(A, E, \mathit{lvl}_E) \qquad \mathit{lvl}_E \leq \mathcal{L}(x)}{\mathit{lvl}_\mathsf{A} \vdash \{A\} x := E}\textsc{LAsgTy}$$

$$\frac{}{\mathit{lvl}_\mathsf{A} \vdash \{A\}\textbf{acquire}(\ell)}\textsc{AcqTy} \qquad \frac{\forall \sigma.\ \sigma \vDash A \implies \mathcal{D}(\mathcal{L}(E), \mathcal{L}(x), \sigma, \lfloor E \rfloor_{\mathit{mem}_\sigma}, \{A\} x \mathrel{\widehat{:=}} E)}{\mathit{lvl}_\mathsf{A} \vdash \{A\} x \mathrel{\widehat{:=}} E}\textsc{DAsgTy}$$

$$\frac{}{\mathit{lvl}_\mathsf{A} \vdash \{A\}\textbf{release}(\ell)}\textsc{RelTy} \qquad \frac{\forall \sigma.\ \sigma \vDash A \implies \mathcal{D}(\mathcal{L}(E), \mathit{lvl}, \sigma, \lfloor E \rfloor_{\mathit{mem}_\sigma}, \{A\} \mathit{lvl} \mathrel{\widehat{!}} E)}{\mathit{lvl}_\mathsf{A} \vdash \{A\} \mathit{lvl} \mathrel{\widehat{!}} E}\textsc{DOutTy}$$

$$\frac{\mathit{lvl}_\mathsf{A} \vdash c_1 \qquad \mathit{lvl}_\mathsf{A} \vdash c_2 \qquad \neg \mathit{sensitivity}(A, E, \mathit{lvl}_\mathsf{A}) \implies \forall i.\ (i, c_1) \overset{\mathit{lvl}_\mathsf{A}}{\sim} (i, c_2)}{\mathit{lvl}_\mathsf{A} \vdash \{A\}\textbf{if } E\ c_1 \textbf{ else } c_2 \textbf{ endif}}\textsc{IfTy}$$

$$\frac{\mathit{lvl} \leq \mathit{lvl}_\mathsf{A} \implies \mathit{sensitivity}(A, E, \mathit{lvl}_\mathsf{A})}{\mathit{lvl}_\mathsf{A} \vdash \{A\} \mathit{lvl}\,!\,E}\textsc{OutTy} \qquad \frac{\mathcal{L}(x) \text{ is undefined}}{\mathit{lvl}_\mathsf{A} \vdash \{A\} x \leftarrow \mathit{lvl}}\textsc{UInTy} \qquad \frac{\mathit{lvl} \leq \mathcal{L}(x)}{\mathit{lvl}_\mathsf{A} \vdash \{A\} x \leftarrow \mathit{lvl}}\textsc{LInTy}$$

$$\frac{\mathit{lvl}_\mathsf{A} \vdash c \qquad \mathit{sensitivity}(A, E, \mathit{lvl}_\mathsf{A}) \qquad \mathit{sensitivity}(I, E, \mathit{lvl}_\mathsf{A})}{\mathit{lvl}_\mathsf{A} \vdash \{A\}\textbf{while } E \textbf{ inv}\{I\}\textbf{do } c}\textsc{WhileTy}$$

Figure 4: Rules of the VERONICA logic.

$\{A_{22}\}$ $\mathit{inmode} := \mathit{inmode} + 1;$
$\{A_{23}\}$ $\mathit{outmode} := \mathit{inmode};$
$\{A_{24}\}$ **release**(ℓ)

Proving that this 5-thread program ls satisfies this policy is relatively straightforward using VERONICA. We employ VERONICA's Owicki-Gries implementation to prove that it satisfies its annotations: $\vDash \mathit{ls}$. We then use the delimited release encoding (Definition 3.4.4) to generate the VERONICA declassification policy $\mathcal{D}$ that encodes the delimited release policy. Next, we use the rules of the VERONICA logic to compositionally prove that each thread ls_i is secure for an arbitrary security level lvl: $\mathit{lvl} \vdash \mathit{ls}_i$. From this proof, since we never use the part of the IFTY rule for branching on secrets, it follows that the program is free of $\mathcal{E}$-secret branching (we prove this result in general in our Isabelle formalisation). Then, by the soundness theorem (Theorem 5.3.1) the program satisfies VERONICA's system security property $\mathit{syssec}_{\mathit{lvl}}(\mathit{ls})$ for arbitrary lvl. Finally, by the delimited release embedding theorem (Theorem 3.4.1) it satisfies its delimited release policy $\mathcal{E}$.

6.2. Confirmed Declassification

Besides delimited release-style policies, VERONICA is geared to verifying state-dependent declassification policies. Such policies are common in systems in which interactions with trusted users authorise declassification decisions. For example, in a sandboxed desktop operating system like Qubes OS [64], a user can copy sensitive files from a protected domain into a less protected one, via an explicit dialogue that requires the user to *confirm* the release of the sensitive information. Indeed, user interactions to make explicit (e.g. "Application X wants permission to access your microphone…") or implicit [65] information access decisions are common in modern computer systems. Yet verifying that concurrent programs only allow information access after successful user confirmation has remained out of reach for prior logics. We show how VERONICA supports such by considering a modification to Figure 1.

Specifically, suppose the thread in Figure 1e is replaced by the one in Figure 5 (left). Instead of using the signature check function CK to decide whether to declassify the $\top$ input, it now asks the user by first outputting the value to be declassified on channel $\top$ and then receiving from the user a boolean response on channel $\bot$.

Naturally the user is trusted, so it is appropriate for their response to this question to be received on the $\bot$ channel. Recall that $\bot$ here means that the information has minimal secrecy, not minimal integrity. Also, since the threat model of Section 3.2 forbids the attacker from supplying channel inputs, we can safely trust the integrity of the user response.

The declassification policy is then specified (see Figure 5 right) as a VERONICA runtime state-dependent declassification predicate $\mathcal{D}$. This predicate specifies that at all times, the most recent output sent (to the user to confirm) on the $\top$ channel is allowed to be declassified precisely when the most recent input consumed from the $\bot$ channel is 1.

A complete formal statement of the policy satisfied by this example is relegated to the extended version of this paper [44] since it is a trivial unfolding of Definition 3.2.8. The resulting property is purely extensional, since $\mathcal{D}$ above refers only to the program's input/output behaviour.

Proving the modified concurrent program secure proceeds similarly as for Section 6.1.This example demonstrates VERONICA's advantages over contemporary logics like COVERN [10] and SECCSL [28], which cannot handle declassification. The example mimics the software functionality of the Cross Domain Desktop Com-

$\{A_{25}\}$ **acquire**(ℓ);
$\{A_{26}\}$ **if** $valid = 1$
  $\{A_{27}\}$ **if** $outmode = 0$
    $\{A_{28}\}$ $\bot buf := buf$
  else
    $\{A_{29}\}$ $\top buf := buf$;
    $\{A_{30}\}$ $\top \,!\, buf$;
    $\{A_{31}\}$ $answer \leftarrow \bot$;
    $\{A_{32}\}$ **if** $answer = 1$
      $\{A_{33}\}$ $\bot buf \;\widehat{:=}\; \top buf$
    endif
  endif
endif ;
$\{A_{34}\}$ **release**(ℓ)

$$\mathcal{D}(lvl_{src}, lvl_{dst}, \sigma, v, E) = (v = lastoutput(\top, \sigma) \wedge lastinput(\bot, \sigma) = 1)$$

Figure 5: User-confirmed declassification.

positor [66] (CDDC), to which these logics were recently applied [10], [28], but—crucially—includes the addition of the CDDC's confirmed-cut-and-paste declassification functionality, which they cannot verify.

6.3. Running Average

As a final stateful-declassification example, consider Figure 6. The top-left thread inputs $\top$-sensitive numbers into the ($\top$-labelled) variable buf and keeps a running sum of the values seem so far in the ($\top$-labelled) variable sum, as well as counting the number of such values consumed in the ($\bot$-labelled) variable cnt. The security policy allows the average of the $\top$ inputs consumed to be declassified so long as the program has consumed more inputs than whatever *threshold* is stored in the ($\bot$-labelled) variable min.

$$\begin{array}{l}\mathcal{D}(lvl_{src}, lvl_{dst}, \sigma, v, E) = \\ \quad \textbf{if } len(inputs(\top, \sigma)) \geq mem_\sigma(min) \textbf{ then} \\ \quad\quad v = avg(inputs(\top, \sigma)) \textbf{ else } \mathsf{false}\end{array}$$

Here the function $inputs(lvl, \sigma)$ extracts from tr_σ all inputs consumed so far on channel lvl.

The threshold min is dynamically updated by the bottom-left thread; the right thread does the declassification.

Thus this system implements a dynamic declassification policy whose enforcement requires careful coordination between the three threads. While its proof is similar to the previous to examples, the declassification policy of this example refers to internal program state (the variable min), so is not extensional, unlike the prior examples. Our Isabelle formalisation contains a modified version of this example that satisfies an extensional policy in which the dynamic threshold is given by the $\bot$ inputs received.

7. Related Work

VERONICA targets compositional and precise verification of expressive forms of information flow security for shared-memory concurrent programs, by decoupled functional correctness (DFC). Prior techniques typically trade precision for expressiveness or vice-versa, or depart from realistic attacker models altogether [67].

$\{A_{35}\}$ **acquire**(ℓavg);
$\{A_{36}\}$ $buf \leftarrow \top$;
$\{A_{37}\}$ $cnt := cnt + 1$;
$\{A_{38}\}$ $sum := sum + buf$;
$\{A_{39}\}$ **release**(ℓavg)

(a) Computing a running sum.

$\{A_{40}\}$ **acquire**(ℓmin);
$\{A_{41}\}$ $min := min + 1$;
$\{A_{42}\}$ **release**(ℓmin)

(b) Increasing the minimum threshold.

$\{A_{43}\}$ **acquire**(ℓavg);
$\{A_{44}\}$ **acquire**(ℓmin);
$\{A_{45}\}$ **if** $cnt > min$
  $\{A_{46}\}$ **if** $cnt > 0$
    $\{A_{47}\}$ $\bot \widehat{!} (sum/cnt)$
  endif
endif ;
$\{A_{48}\}$ **release**(ℓmin);
$\{A_{49}\}$ **release**(ℓavg)

(c) Declassifying the average.

Figure 6: Declassifying the average with dynamic threshold.

Two recent logics that favour precision over expresiveness are COVERN [10] and SECCSL [28]. VERONICA is more expressive than COVERN [10] and SECCSL [28], because of VERONICA's ability to reasoning about declassification, which these prior logics cannot handle.

The VERONICA logic is arguably much simpler than COVERN, even while being more expressive. Like VERONICA, SECCSL is also relatively simple and clean. However, it focuses on automated application via symbolic execution [28]. VERONICA, in contrast, was designed to favour precise reasoning over automation.

VERONICA borrows its functional correctness annotations from the Owicki-Gries proof technique [50]. In doing so, it inherits a well known [68] property of that technique: the need to sometimes introduce *ghost variables* when reasoning about certain concurrent programs. Such an example is depicted in the extended version of this paper [44].

Karbyshev et al. [29] present a highly precise separation logic based method for compositionally proving security of concurrent programs. Unlike VERONICA, their approach supports a far more flexible scheduler model, including reasoning about benign races on public variables, dynamic thread creation and thread→scheduler interactions. Unlike VERONICA, [29] doesn't support declassification.

Others have examined information flow verification for *distributed* concurrent programs, in which threads do not share memory. Bauereiß et al. [27] present a method for verifying the security of such programs, including for some declassification policies and apply it to verify the key functionality of a distributed social media platform. Li et al. [33] present a rely-guarantee based method, tailored to systems in which the *presence* of messages on a channel can reveal sensitive information. In VERONICA, input is always assumed to be available on all channels.

Decoupled functional correctness was foreshadowed in the recent work of Li and Zhang [69] (as well as in aspects of Amtoft et al. [7]). Li and Zhang's approach supports relatively precise reasoning about data-dependent sensitivity of sequential (i.e. non-concurrent) programs that carry annotations on assignment statements. VERONICA extends this idea across the entire program and applies it to compositional reasoning about shared-memory concurrent programs.

Relational decomposition [70], [71] and the *product program* approaches [72]–[74] encode security reasoning via functional correctness. Instead VERONICA exploits compositional functional correctness to aid security reasoning.

8. Conclusion

We presented VERONICA, the first compositional logic for proving information flow security for shared-memory concurrent programs that supports precise reasoning about expressive security policies. It embodies a new approach to building such logics: decoupled functional correctness.

As we demonstrated, VERONICA supports reasoning about myriad security policies, including delimited release-style declassification, value-dependent sensitivity and runtime-state dependent declassification, and their cooperative enforcement via non-trivial thread interactions. VERONICA sets a new standard for reasoning methods for concurrent information flow security.

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

References

[1] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong, "Precise, dynamic information flow for database-backed applications," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, vol. 51, no. 6, 2016, pp. 631–647.

[2] L. Zheng and A. C. Myers, "Dynamic security labels and static information flow control," *International Journal of Information Security*, vol. 6, no. 2–3, Mar. 2007.

[3] N. Swamy, J. Chen, and R. Chugh, "Enforcing stateful authorization and information flow policies in Fine," in *European Symposium on Programming (ESOP)*, March 2010.

[4] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang, "Secure distributed programming with value-dependent types," in *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2011, pp. 266–278.

[5] L. Lourenço and L. Caires, "Dependent information flow types," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Mumbai, India, Jan. 2015, pp. 317–328.

[6] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[7] T. Amtoft, S. Bandhakavi, and A. Banerjee, "A logic for information flow in object-oriented programs," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2006, pp. 91–102.

[8] T. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein, "Noninterference for operating system kernels," in *International Conference on Certified Programs and Proofs (CPP)*, Dec. 2012, pp. 126–142.

[9] A. Nanevski, A. Banerjee, and D. Garg, "Verification of information flow and access control policies with dependent types," in *IEEE Symposium on Security & Privacy (S&P)*, May 2011, pp. 165–179.

[10] T. Murray, R. Sison, and K. Engelhardt, "COVERN: A logic for compositional verification of information flow control," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, London, United Kingdom, Apr. 2018.

[11] N. Broberg and D. Sands, "Paralocks: role-based information flow control and beyond," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, vol. 45, no. 1, 2010, pp. 431–444.

[12] B. van Delft, S. Hunt, and D. Sands, "Very static enforcement of dynamic policies," in *International Conference on Principles of Security and Trust (POST)*, 2015, pp. 32–52.

[13] N. Broberg, B. van Delft, and D. Sands, "The anatomy and facets of dynamic policies," in *IEEE Computer Security Foundations Symposium (CSF)*, 2015, pp. 122–136.

[14] A. Askarov and S. Chong, "Learning is change in knowledge: Knowledge-based security for dynamic policies," in *IEEE Computer Security Foundations Symposium (CSF)*, 2012, pp. 308–322.

[15] C. Zhang, "Conditional information flow policies and unwinding relations," in *International Symposium on Trustworthy Global Computing (TGC)*, 2011, pp. 227–241.

[16] S. Eggert and R. van der Meyden, "Dynamic intransitive noninterference revisited," *Formal Aspects of Computing*, vol. 29, no. 6, pp. 1087–1120, 2017.

[17] D. McCullough, "Specifications for multi-level security and a hook-up," in *IEEE Symposium on Security & Privacy (S&P)*, 1987, pp. 161–161.

[18] G. Smith and D. Volpano, "Secure information flow in a multi-threaded imperative language," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1998, pp. 355–364.

[19] T. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah, "Compositional verification and refinement of concurrent value-dependent noninterference," in *IEEE Computer Security Foundations Symposium (CSF)*, Jun. 2016, pp. 417–431.

[20] D. Volpano and G. Smith, "Probabilistic noninterference in a concurrent language," *Journal of Computer Security*, vol. 7, no. 2,3, pp. 231–253, 1999.

[21] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multi-threaded programs," in *IEEE Computer Security Foundations Workshop (CSFW)*, 2000, pp. 200–215.

[22] G. R. Andrews and R. P. Reitman, "An axiomatic approach to information flow in parallel programs," Cornell University, Tech. Rep., 1978.

[23] ——, "An axiomatic approach to information flow in programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 2, no. 1, pp. 56–76, 1980.

[24] H. Mantel and A. Reinhard, "Controlling the What and Where of declassification in language-based security," in *European Symposium on Programming (ESOP)*, 2007, pp. 141–156.

[25] A. Bossi, C. Piazza, and S. Rossi, "Compositional information flow security for concurrent programs," *Journal of Computer Security*, vol. 15, no. 3, pp. 373–416, 2007.

[26] A. Lux, H. Mantel, and M. Perner, "Scheduler-independent declassification," in *International Conference on Mathematics of Program Construction*, 2012, pp. 25–47.

[27] T. Bauereiß, A. P. Gritti, A. Popescu, and F. Raimondi, "CoSMeDis: a distributed social media platform with formally verified confidentiality guarantees," in *IEEE Symposium on Security & Privacy (S&P)*, 2017, pp. 729–748.

[28] G. Ernst and T. Murray, "SECCSL: Security Concurrent Separation Logic," in *International Conference on Computer Aided Verification (CAV)*. Springer, 2019, pp. 208–230.

[29] A. Karbyshev, K. Svendsen, A. Askarov, and L. Birkedal, "Compositional non-interference for concurrent programs via separation and framing," in *International Conference on Principles of Security and Trust (POST)*, 2018.

[30] H. Mantel, D. Sands, and H. Sudbrock, "Assumptions and guarantees for compositional noninterference," in *IEEE Computer Security Foundations Symposium (CSF)*, Cernay-la-Ville, France, Jun 2011, pp. 218–232.

[31] H. R. Nielson, F. Nielson, and X. Li, "Hoare logic for disjunctive information flow," in *Programming Languages with Applications to Biology and Security*, 2015, pp. 47–65.

[32] H. R. Nielson and F. Nielson, "Content dependent information flow control," *Journal of Logical and Algebraic Methods in Programming*, vol. 87, pp. 6–32, 2017.

[33] X. Li, H. Mantel, and M. Tasch, "Taming message-passing communication in compositional reasoning about confidentiality," in *Asian Symposium on Programming Languages and Systems (APLAS)*, 2017, pp. 45–66.

[34] J. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security & Privacy (S&P)*, Apr 1982, pp. 11–20.

[35] C. B. Jones, "Development methods for computer programs including a notion of interference," D.Phil. thesis, University of Oxford, Jun. 1981.

[36] J. McLean, "A general theory of composition for trace sets closed under selective interleaving functions," in *IEEE Symposium on Security & Privacy (S&P)*, 1994, pp. 79–93.

[37] A. Zakinthinos and E. S. Lee, "A general theory of security properties," in *IEEE Symposium on Security & Privacy (S&P)*, 1997, pp. 94–102.

[38] H. Mantel, "On the composition of secure systems," in *IEEE Symposium on Security & Privacy (S&P)*, 2002, pp. 88–101.

[39] W. Rafnsson and A. Sabelfeld, "Compositional information-flow security for interactive systems," in *IEEE Computer Security Foundations Symposium (CSF)*, 2014, pp. 277–292.

[40] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2-3, pp. 167–187, 1996.

[41] T. Amtoft and A. Banerjee, "Information flow analysis in logical form," in *Static Analysis Symposium (SAS)*, 2004, pp. 100–115.

[42] T. Nipkow, L. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2283.

[43] D. Schoepe, T. Murray, and A. Sabelfeld, "VERONICA: Isabelle theories," https://bitbucket.org/covern/veronica, 2020.

[44] ——, "VERONICA: Expressive and precise concurrent information flow security (extended version with technical appendices)," arXiv preprint arXiv:2001.11142 [cs.LO], 2020, https://arxiv.org/abs/2001.11142.

[45] A. Sabelfeld and A. C. Myers, "A model for delimited information release," in *International Symposium on Software Security-Theories and Systems (ISSS)*, 2003, pp. 174–191.

[46] A. Banerjee, D. A. Naumann, and S. Rosenberg, "Expressive declassification policies and modular static enforcement," in *IEEE Symposium on Security & Privacy (S&P)*, 2008, pp. 339–353.

[47] N. Khakpour, O. Schwarz, and M. Dam, "Machine assisted proof of armv7 instruction level isolation properties," in *International Conference on Certified Programs and Proofs (CPP)*, 2013, pp. 276–291.

[48] A. Ferraiuolo, W. Hua, A. C. Myers, and G. E. Suh, "Secure information flow verification with mutable dependent types," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, p. 6.

[49] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[50] S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs," *Acta Informatica*, vol. 6, pp. 319–340, 1976.

[51] P. W. O'Hearn, "Resources, concurrency and local reasoning," in *International Conference on Concurrency Theory (CONCUR)*. Springer, 2004, pp. 49–67.

[52] D. E. Bell and L. J. La Padula, "Secure computer system: Unified exposition and Multics interpretation," MITRE Corp., Tech. Rep. MTR-2997, Mar. 1976.

[53] L. Prensa Nieto and J. Esparza, "Verifying single- and multi-mutator garbage collectors with Owicki/Gries in Isabelle/HOL," in *Mathematical Foundations of Computer Science (MFCS)*, ser. Lecture Notes in Computer Science, vol. 1893, 2000, pp. 619–628.

[54] L. Prensa Nieto, "Verification of parallel programs with the Owicki-Gries and rely-guarantee methods in Isabelle/HOL," Ph.D. dissertation, Technische Universität München, 2002.

[55] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A practical system for verifying concurrent C," in *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, ser. Lecture Notes in Computer Science, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., vol. 5674. Springer, 2009, pp. 23–42. [Online]. Available: https://doi.org/10.1007/978-3-642-03359-9_2

[56] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *IEEE Symposium on Security & Privacy (S&P)*, 2007, pp. 207–221.

[57] N. Broberg and D. Sands, "Flow-sensitive semantics for dynamic information flow policies," in *ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, 2009, pp. 101–112.

[58] T. Murray and P. C. van Oorschot, "BP: Formal proofs, the fine print and side effects," in *IEEE Cybersecurity Development Conference (SecDev)*, 2018, to appear.

[59] E. Cecchetti, A. C. Myers, and O. Arden, "Nonmalleable information flow control," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 1875–1891.

[60] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: from general purpose to a proof of information flow enforcement," in *IEEE Symposium on Security & Privacy (S&P)*, May 2013, pp. 415–429.

[61] D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières, "Eliminating cache-based timing attacks with instruction-based scheduling," in *European Symposium on Research in Computer Security (ESORICS)*, Sep 2013, pp. 718–735.

[62] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 17, no. 5, pp. 517–548, 2009.

[63] R. W. Floyd, "Assigning meanings to programs," *Mathematical Aspects of Computer Science*, vol. 19, pp. 19–32, 1967.

[64] J. Rutkowska and R. Wojtczuk, "Qubes OS architecture," https://www.qubes-os.org/attachment/wiki/QubesArchitecture/arch-spec-0.3.pdf, Invisible Things Lab, Tech. Rep., Jan. 2010.

[65] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan, "User-driven access control: Rethinking permission granting in modern operating systems," in *IEEE Symposium on Security & Privacy (S&P)*, 2012, pp. 224–238.

[66] M. Beaumont, J. McCarthy, and T. Murray, "The cross domain desktop compositor: using hardware-based video compositing for a multi-level secure user interface," in *Annual Computer Security Applications Conference (ACSAC)*. ACM, 2016, pp. 533–545.

[67] I. Bastys, F. Piessens, and A. Sabelfeld, "Prudent design principles for information flow control," in *ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, Oct. 2018.

[68] A. Malkis and L. Mauborgne, "On the strength of Owicki-Gries for resources," in *Asian Symposium on Programming Languages and Systems (APLAS)*. Springer, 2011, pp. 172–187.

[69] P. Li and D. Zhang, "Towards a flow-and path-sensitive information flow analysis," in *IEEE Computer Security Foundations Symposium (CSF)*, 2017, pp. 53–67.

[70] L. Beringer and M. Hofmann, "Secure information flow and program logics," in *IEEE Computer Security Foundations Symposium (CSF)*, 2007, pp. 233–248.

[71] L. Beringer, "Relational decomposition," in *International Conference on Interactive Theorem Proving (ITP)*, 2011, pp. 39–54.

[72] Á. Darvas, R. Hähnle, and D. Sands, "A theorem proving approach to analysis of secure information flow," in *International Conference on Security in Pervasive Computing*. Springer, 2005, pp. 193–209.

[73] G. Barthe, P. R. D'argenio, and T. Rezk, "Secure information flow by self-composition," *Mathematical Structures in Computer Science*, vol. 21, no. 6, pp. 1207–1252, 2011.

[74] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *Static Analysis Symposium (SAS)*, 2005, pp. 352–367.

Appendix

1. Ancillary Definitions

Definition A.1 (State Agreement under $\mathcal{E}$ (full definition))**.** *We say that two states σ and σ' agree under the delimited release policy $\mathcal{E}$ for an attacker at level lvl_{A}, written $\sigma \overset{lvl_{\mathsf{A}},\mathcal{E}}{\approx} \sigma'$, iff (1) $\sigma \overset{lvl_{\mathsf{A}}}{\approx} \sigma$, (2) their memories agree on all variables, (3) the same number of inputs has been consumed so far in each, and (4) the environment obtained by appending the inputs consumed so far in σ to env_σ agrees under $\mathcal{E}$ with the environment obtained by doing likewise to σ'.*

Here, condition (2) is encodes the simplifying assumption that the initial memories contain no secrets. Conditions (3) and (4) are more complicated than might be expected due to having generalised over all σ: for initial states σ and σ' in which no events have been been performed (i.e. tr_σ and $tr_{\sigma'}$ are both empty), condition (3) holds trivially and condition (4) collapses to $env_\sigma \overset{lvl_{\mathsf{A}},\mathcal{E}}{\approx} env_{\sigma'}$: agreement of the two environments under $\mathcal{E}$. In this way this more general definition is morally equivalent to the simpler one (Definition 3.4.2) of Section 3.4.

Definition A.2 (Absence of $\mathcal{E}$-Secret Branching)**.** *We say that a program ls* doesn't branch on secrets that the delimited release policy $\mathcal{E}$ forbids from releasing*, when observed by an attacker at level lvl_{A}, when if for all schedules $sched$ and initial states σ, if the program executes to some configuration y, then that execution can be matched from any other initial state σ' for which $\sigma \overset{lvl_{\mathsf{A}},\mathcal{E}}{\approx} \sigma'$ to reach a configuration y' whose thread local states $ls_{y'}$ is equal to ls_y, the thread local states of y (meaning that the two runs are still executing the same code in all threads) and, moreover, the same number of lvl_{A}-visible events have been performed so far in y and y' and, for all levels lvl, the same number of inputs from channel lvl has been consumed in both y and y'.*

lvl_{A}-Bisimilarity

lvl_{A}-bisimilarity is defined via the notion of an lvl_{A}-*secure bisimulation*. Essentially a lvl_{A}-secure bisimulation is a relational invariant on the execution of a thread that ensures that each step of its execution satisfies what we call lvl_{A}-*step security*.

Definition A.3 (lvl-Step Security)**.** *Let lvl be a security level. Let σ and σ_2 be global states such that a single execution step has occurred from σ to reach σ_2, and let σ' and σ'_2 be likewise, such that $\sigma \overset{lvl}{\approx} \sigma'$. Then these states satisfy lvl-step security, written $stepsec_{lvl}(\sigma, \sigma_2, \sigma', \sigma'_2)$, iff:*

- *If the execution step from σ produced a declassification event visible at level lvl, then, whenever the same event is produced by the step from σ', we require that $\sigma_2 \overset{lvl}{\approx} \sigma'_2$.*
- *If the execution step from σ produced a declassification event not visible at level lvl, then we require that $\sigma_2 \overset{lvl}{\approx} \sigma'_2$ unconditionally.*
- *In either case, the number of lvl-visible events in tr_{σ_2} and $tr_{\sigma'_2}$ must be equal.*
- *Otherwise, if no declassification event is produced in the step from σ, we require that $\sigma_2 \overset{lvl}{\approx} \sigma'_2$.*

Definition A.4 (lvl-Secure Bisimulation)**.** *For a security level lvl, a binary relation $\mathcal{R}$ on thread local states (i, c) is an lvl-secure bisimulation iff whenever $(i, c)\ \mathcal{R}\ (i', c')$:*

- $i = i'$
- $c = \textbf{\textit{stop}} \iff c' = \textbf{\textit{stop}}$
- *An execution step of $((i, c), \sigma) \rightsquigarrow ((i, c_2), \sigma_2)$ from a global state σ that satisfies c's annotation, can be matched by a step $((i, c'), \sigma') \rightsquigarrow ((i, c'_2), \sigma'_2)$ from any global state σ' that satisfies c''s annotation whenever $\sigma \overset{lvl}{\approx} \sigma'$. Moreover, in that case $(i, c_2)\ \mathcal{R}\ (i, c'_2)$ is preserved and lvl-step security is satisfied: $stepsec_{lvl}(\sigma, \sigma_2, \sigma', \sigma'_2)$.*

Definition A.5 (lvl-Bisimilarity)**.** *We say that two local thread states (i, c) and (i', c') are lvl-bisimilar, written $(i, c) \overset{lvl}{\sim} (i', c')$ whenever there exists a lvl-secure bisimulation $\mathcal{R}$ that relates them: $(i, c)\ \mathcal{R}\ (i', c')$.*

$$\frac{(ls_i, \sigma) \rightsquigarrow (ls_i', \sigma')}{(ls_0, \ldots, ls_i, \ldots, ls_{n-1}, \sigma, i \cdot sched') \to (ls_0, \ldots, ls_i', \ldots, ls_{n-1}, \sigma', sched')}\textsc{GStep}$$

$$\frac{\nexists y.\ (ls_i, \sigma) \rightsquigarrow y}{(ls_0, \ldots, ls_i, \ldots, ls_{n-1}, \sigma, i \cdot sched') \to (ls_0, \ldots, ls_i, \ldots, ls_{n-1}, \sigma, sched')}\textsc{GWait}$$

Figure 7: Concurrent execution. Here, $\cdot \rightsquigarrow \cdot$ is the small-step semantics of individual thread programs (see Figure 8).

$$\frac{\lfloor E \rfloor_{mem_\sigma} = v \qquad mem' = mem_\sigma[x \mapsto v]}{((i, \{A\} x := E), (env_\sigma, mem_\sigma, locks_\sigma, tr_\sigma)) \rightsquigarrow ((i, \textbf{stop}), (env_\sigma, mem', locks_\sigma, tr_\sigma))}\textsc{Assign}$$

$$\frac{\lfloor E \rfloor_{mem_\sigma} = v \qquad mem' = mem_\sigma[x \mapsto v] \qquad tr' = tr_\sigma \cdot \mathbf{d}\langle \mathcal{L}(x), v, E \rangle}{((i, \{A\} x \mathrel{\widehat{:=}} E), (env_\sigma, mem_\sigma, locks_\sigma, tr_\sigma)) \rightsquigarrow ((i, \textbf{stop}), (env_\sigma, mem', locks_\sigma, tr'))}\textsc{DAssign}$$

$$\frac{\lfloor E \rfloor_{mem_\sigma} = v \qquad tr' = tr_\sigma \cdot \mathbf{out}\langle lvl, v, E \rangle}{((i, \{A\} lvl\, !\, E), (env_\sigma, mem_\sigma, locks_\sigma, tr_\sigma)) \rightsquigarrow ((i, \textbf{stop}), (env_\sigma, mem_\sigma, locks_\sigma, tr'))}\textsc{Output}$$

$$\frac{\lfloor E \rfloor_{mem_\sigma} = v \qquad tr' = tr_\sigma \cdot \mathbf{d}\langle lvl, v, E \rangle}{((i, \{A\} lvl\, \widehat{!}\, E), (env_\sigma, mem_\sigma, locks_\sigma, tr_\sigma)) \rightsquigarrow ((i, \textbf{stop}), (env_\sigma, mem_\sigma, locks_\sigma, tr'))}\textsc{DOutput}$$

$$\frac{env_\sigma(lvl) = v \cdot vs \qquad env' = env_\sigma[lvl \mapsto vs] \qquad mem' = mem_\sigma[x \mapsto v] \qquad tr' = tr_\sigma \cdot \mathbf{in}\langle lvl, v \rangle}{((i, \{A\} lvl \leftarrow v), (env_\sigma, mem_\sigma, locks_\sigma, tr_\sigma)) \rightsquigarrow ((i, \textbf{stop}), (env', mem', locks_\sigma, tr'))}\textsc{Input}$$

$$\frac{\lfloor E \rfloor_{mem_\sigma} = \mathsf{true}}{((i, \{A\} \textbf{if } E\ c_1 \textbf{ else } c_2 \textbf{ endif}), \sigma) \rightsquigarrow ((i, c_1), \sigma)}\textsc{IfT} \qquad \frac{\lfloor E \rfloor_{mem_\sigma} \neq \mathsf{true}}{((i, \{A\} \textbf{if } E\ c_1 \textbf{ else } c_2 \textbf{ endif}), \sigma) \rightsquigarrow ((i, c_2), \sigma)}\textsc{IfF}$$

$$\frac{((i, c_1), \sigma) \rightsquigarrow ((i, c_1'), \sigma') \qquad c_1' \neq \textbf{stop}}{((i, c_1;\ c_2), \sigma) \rightsquigarrow ((i, c_1';\ c_2), \sigma')}\textsc{Seq} \qquad \frac{((i, c_1), \sigma) \rightsquigarrow ((i, c_1'), \sigma') \qquad c_1' = \textbf{stop}}{((i, c_1;\ c_2), \sigma) \rightsquigarrow ((i, c_2), \sigma')}\textsc{SeqStop}$$

$$\frac{\lfloor E \rfloor_{mem_\sigma} = \mathsf{true}}{((i, \{A\} \textbf{while } E \textbf{ inv}\{I\} \textbf{do } c), \sigma) \rightsquigarrow ((i, c;\ \{A\} \textbf{while } E \textbf{ inv}\{I\} \textbf{do } c), \sigma)}\textsc{WhileT}$$

$$\frac{\lfloor E \rfloor_{mem_\sigma} \neq \mathsf{true}}{((i, \{A\} \textbf{while } E \textbf{ inv}\{I\} \textbf{do } c), \sigma) \rightsquigarrow ((i, \textbf{stop}), \sigma)}\textsc{WhileF}$$

$$\frac{locks_\sigma(\ell) \text{ is undefined} \qquad locks' = locks_\sigma[\ell \mapsto i]}{((i, \{A\} \textbf{acquire}(\ell)), (env_\sigma, mem_\sigma, locks_\sigma, tr_\sigma)) \rightsquigarrow ((i, \textbf{stop}), (env_\sigma, mem_\sigma, locks', tr_\sigma))}\textsc{Acquire}$$

$$\frac{locks_\sigma(\ell) = i \qquad locks' = locks_\sigma[\ell \text{ is undefined}]}{((i, \{A\} \textbf{release}(\ell)), (env_\sigma, mem_\sigma, locks_\sigma, tr_\sigma)) \rightsquigarrow ((i, \textbf{stop}), (env_\sigma, mem_\sigma, locks', tr_\sigma))}\textsc{Release}$$

Figure 8: Semantics of threads, where $\sigma = (env_\sigma, mem_\sigma, locks_\sigma, tr_\sigma)$.