Compositional Verification and Refinement of Concurrent Value-Dependent Noninterference

Toby Murray*†, Robert Sison†, Edward Pierzchalski†, Christine Rizkallah†
*University of Melbourne †Data61, NICTA & University of New South Wales
firstname.lastname@unimelb.edu.au firstname.lastname@data61.csiro.au

***Abstract*—Value-dependent noninterference allows the classification of program variables to depend on the contents of other variables, and therefore is able to express a range of data-dependent security policies. However, so far its static enforcement mechanisms for software have been limited either to progress- and termination-insensitive noninterference for sequential languages, or to concurrent message-passing programs without shared memory. Additionally, there exists no methodology for preserving value-dependent noninterference for shared memory programs under compositional refinement. This paper presents a flow-sensitive dependent type system for enforcing timing-sensitive value-dependent noninterference for shared memory concurrent programs, comprising a collection of sequential components, as well as a compositional refinement theory for preserving this property under componentwise refinement. Our results are mechanised in Isabelle/HOL.**

I. Introduction

Few foundational topics of computer security have received more study than information flow security, which deals with the problem of preventing unwanted information leakage. Under the traditional view of *language-based noninterference* [SM03], each program variable is assigned a fixed security *classification*, which identifies the kind of data it contains. Noninterference holds when information derived from confidential variables cannot be inferred by observing changes to non-confidential ones. *Timing-sensitive noninterference* requires that both *how* and *when* the contents of non-confidential variables change is independent of the contents of confidential variables, and so no information about confidential variables is leaked.

Value-dependent noninterference allows the security classification of variables to depend on the contents of other program variables. Its static enforcement has been studied primarily in the context of dependent (security) type systems [ZM07], [SCC10], [SCF+11], [LC15], [ZWSM15], relational program logics [ABB06], [MMB+12], and combinations thereof [NBG11].

The aforementioned work has clearly established the utility of value-dependent noninterference for enforcing a range of (state-dependent) security policies, besides declassification and erasure [SBN13]. However, all prior work on its static enforcement for software has been limited either to sequential programs with progress- and timing-insensitive noninterference, or (just recently) to process-algebra style formalisms [LNNF16] without shared memory, where compositionality is relatively straightforward. Little work has examined the compositional verification of value-dependent noninterference for concurrent shared-memory programs, despite it being an active area of research for traditional (non-value-dependent) noninterference [VS99], [SS00], [BC02], [MSS11].

In addition to verifying the security of a program against its source level semantics, true assurance requires verifying that a program's implementation (e.g. after compilation) is also secure. This is particularly challenging for timing-sensitive noninterference, which requires an implementation's timing behaviour to be independent of confidential data, especially when source language semantics typically abstract over the execution time of program statements. This problem is even worse for concurrent shared-memory programs, which comprise a set of concurrently-executing sequential *components*. Here, the correct compilation of one component necessarily depends on implicit assumptions made about the absence of unwanted effects caused by other concurrently-running components. There exists no theory allowing one to transfer a compositional proof of timing-sensitive noninterference for a concurrent program to its refined implementation, one component-at-a-time.

This paper addresses these shortcomings via two main contributions, of which Sec. II provides an overview. These contributions are constructed on the value-dependent extension [Mur15] of the compositional theory of timing-sensitive, concurrent noninterference of [MSS11], summarised in Sec. III.

In Sec. IV, we present for a simple imperative language what is, to our knowledge, the first flow-sensitive, dependent type system for compositionally verifying value-dependent noninterference of shared memory concurrent programs, taking into account the assumptions each component makes about the others with which it runs concurrently.

In Sec. V, we present the first compositional refinement theory for preserving timing-sensitive, value-dependent noninterference. We define a suitable notion of refinement for establishing that each component of a concurrent system has been securely refined to its implementation so that, when composed together, the implementations running in parallel are guaranteed secure. This theory introduces the notion of a concrete *coupling invariant* to ensure that "constant-time" parts of the source program remain constant-time in the concrete implementation.

Sec. VI exercises this theory to prove the intuition that programs whose execution never branches on confidential data

(and so are trivially constant-time) can be securely refined by what we call *simple* refinement relations, without a coupling invariant. Such refinement relations must naturally not introduce new branching on confidential data, or choose to refine programs differently based on confidential memory contents.

We discuss related work in Sec. VII before concluding in Sec. VIII. Our results are mechanised in Isabelle/HOL, and were typeset here by Isabelle [NPW02] to avoid transcription errors. The total size of our Isabelle/HOL formalisation (including examples) is over 20,000 lines.

II. Overview

This section gives a high-level overview of the contributions of this paper, described in detail in subsequent sections.

A. Type System

1) Language: The type system we present in Sec. IV is for a tiny imperative language for writing sequential components that run concurrently to one another with access to globally shared memory. As is usual, we restrict our attention to noninterference over the two-point lattice of classifications $\{\mathsf{Low}, \mathsf{High}\}$ ordered by $\leq$ in the natural way. Fig. 1a shows an example component that reads from an input buffer buffer, modelled as a single variable, and copies the contents into one of two output buffers, high-var and low-var, depending on whether the data held in the input buffer was classified High or Low respectively. The classification of the input buffer buffer is value-dependent, and depends on the contents of the *control* variable control: when control is zero, buffer is Low and otherwise it is classified High. control is itself statically classified Low, as required for all such variables on which the classification of other variables depend [Mur15] to prevent information leakage from changes to variable classifications. The variable temp is also statically classified Low.

The comments delimited by /* and */ indicate local *assumptions* made by this fragment of code about the behaviour of other components in the system. These assumptions are what allow us to verify this component as being secure in isolation from the other components in the system that it may execute concurrently with. Specifically, given a system comprising a set of components, to verify the entire system as secure it suffices to (1) verify each component on its own under its local assumptions, and (2) to prove that each component adheres to the assumptions of all others. This result is proved formally in [Mur15], extending [MSS11], which we summarise later in Sec. III. Verifying each component under its local assumptions is precisely the purpose of the type system we present in this paper. Existing techniques can be applied to verify that each component adheres to the assumptions of all others [MMOPW15].

Before executing the first **skip** statement, the code *acquires* the assumption that no other component will modify (**AsmNoW**) control and it operates under that assumption for its entire execution because it never explicitly *releases* it. Before executing the second **skip** statement, it acquires the assumption that no other component will read or modify (**AsmNoRW**) temp, a variable that temporarily holds the contents of buffer while the code is working out what classification buffer is.

This assumption about temp is maintained up until the final **skip** statement, at which it is *released*. The control assumption ensures that the classification of buffer won't change in between this code having read its value and then later examining control. The temp assumption ensures that, in the event that buffer was classified High, that no other component will inadvertently read that High data while it is residing in temp, which would be insecure since temp is classified Low.

2) Typing Contexts and Judgements: The judgements of our flow-sensitive type system are of the form $\vdash \Gamma,\mathcal{S},P\ \{c\}\ \Gamma',\mathcal{S}',P'$ for statements c of our language. Here, Γ, $\mathcal{S}$ and P record information that is true about the state from which c executes and Γ', $\mathcal{S}'$ and P' record updated information that is true as a result of c having executed.

The environment Γ tracks the (dependent) classification of the data in variables throughout the program. The types in Γ depend on memory. To ease its automation, we represent these types by sets of predicates on memory. A type t is *interpreted* as Low in memory *mem* when all predicates in the set t hold for *mem*, and is interpreted as High in *mem* otherwise. With this representation of types, we can phrase subtyping and type-equivalence via predicate entailment, which eases automation. After the assignment $\mathsf{temp} := \mathsf{buffer}$ in the program in Fig. 1a, Γ is updated with the mapping $\mathsf{temp} \mapsto \{\mathsf{control} == 0\}$, the predicate representation of the type (i.e. value-dependent classification) of buffer.

To decide whether an assignment like $\mathsf{low\text{-}var} := \mathsf{temp}$ is secure, our type system maintains a set of predicates P that are known to be true at the current point in the program. When this assignment is encountered in the program in Fig. 1a, P contains the predicate $\mathsf{control} == 0$ because this assignment is performed in the **then**-branch of the program's single **if**-statement. Proving that this assignment is secure then requires showing that the type of temp in Γ is a *subtype* of the type of low-var, *under the predicates in P*. By this we mean the following. Let t be the type of temp in Γ. Then for all memories *mem* satisfying P, the interpretation of t in *mem* is $\leq$ the interpretation of low-var's type (which is statically Low). This requirement is phrased in terms of predicate entailment: whenever all predicates in P hold, we require that all predicates in t also hold, implying that the data in t is Low. Since the sole predicate in t, $\mathsf{control} == 0$, is also present in P, the assignment is straightforwardly secure.

The final component $\mathcal{S}$ tracked by the type system accounts for the reality that other components in the system may modify any variable for which the current component does not have an **AsmNoW** or **AsmNoRW** assumption — i.e. any variable that is assumed writable. $\mathcal{S}$ is simply a pair of sets (NW, NRW) where NW and NRW record, respectively, those variables for which the component has an **AsmNoW** and **AsmNoRW** assumption. It is updated in the natural way when statements like **skip** /* control $+=_m$ **AsmNoW** */ are encountered.

```
skip  /* control +=_m AsmNoW */;
skip  /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip  /* temp -=_m AsmNoRW */
```

(a) Reading a dynamically-classified buffer, adapted from [Mur15].

```
skip  /* control_C +=_m AsmNoW */;
skip  /* temp_C +=_m AsmNoRW */;
temp_C := buffer_C;
reg_C := control_C;
if reg_C == 0 then
  low-var_C := temp_C
else
  high-var_C := temp_C
endif;
temp_C := 0;
skip  /* temp_C -=_m AsmNoRW */
```

(b) A trivial refinement of Fig. 1a.

Fig. 1: Value-dependent noninterference and its refinement.

3) Stability: The purpose of $\mathcal{S}$ is to restrict the information that the type system records in Γ and P to being that which pertains only to *stable* variables, which are those contained in $\mathcal{S}$. It is only stable variables whose values are guaranteed not to be altered by other components, and so it makes sense to track information only about them. Hence, every variable mentioned in Γ must be stable, as must every variable mentioned in a predicate in P. The type system maintains this invariant.

We make the explicit design decision that our type system records information that pertains only to the current state of the program; it does not attempt to record information about the values of variables in the past, for instance. This keeps the set of predicates P from increasing monotonically. However, as a result, all types in Γ need to be *stable* in the sense that the variables that they depend on need to be included in $\mathcal{S}$. This ensures that we can always interpret all dependent types in Γ under the entire set of predicates in P.

4) Context Rewriting: The ability to keep the amount of information that our type system must deal with under control comes at the price of complicating its application in some instances. To see why, suppose we altered Fig. 1a by adding the line **skip** /* control $-=_m$ **AsmNoW** */ directly preceding the assignment to low-var. Doing so releases the assumption that control won't be modified, making it unstable. Thus the type of temp in Γ, which recall is $\{\text{control} == 0\}$, no longer applies. This type refers to the value of control when buffer was read. However, its value may have since changed because it is now unstable.

To solve this problem, the typing rules allow the typing context Γ to be rewritten by replacing types with ones that are equivalent under the set of predicates P. Like subtyping, this equivalence is phrased via predicate entailment, easing its automation. Thus, at the point where control is determined to be zero (and thus P contains the predicate control $== 0$), Γ can be rewritten to replace temp's type with the equivalent one under P of simply the empty set of predicates $\emptyset$, which refers to no variables and so is trivially stable.

5) Automation: Our type system generalises over the program's languages of arithmetic and boolean expressions, and reuses the program's boolean expressions as the language of predicates (in types and in P). Having fixed the expression languages, however, the application of our type system is easily automated. We have implemented a set of automated tactics in Isabelle's Eisbach [MWM14], [MMW16] proof method language to automate typing proofs for programs employing a trivial predicate language. The security proofs of examples like Fig. 1a are entirely automated, for instance. The tactics at present require user intervention to specify when and how to rewrite the typing context Γ, as well as in some instances to provide post-contexts following **if**-statements (as explained later in Sec. IV). However, because type equivalence can be deduced from predicate entailment, automating this reasoning should also be feasible for an appropriately expressive predicate language.

B. Refinement

1) Component Refinement: The primary purpose of our theory of refinement (presented in Sec. V) is to allow a proof of security for a concurrent program, carried out one-component-at-a-time (e.g. using our type system), to be transferred to a more *concrete* semantics for the program with as little work as possible. The more concrete semantics for the concurrent program is one arising from having refined each individual component of the more abstract program individually. A component might be refined by e.g. compiling it and interpreting the object code against an assembly-level semantics, or by manually constructing a more detailed model of the component than that used for the security proof, etc.

Fig. 1b depicts a trivial refinement of the program in Fig. 1a. Each variable that corresponds to a variable from Fig. 1a has the same name as the original variable, but with the subscript "*C*" (for "concrete") appended. Notice that there is a new variable reg_C that has no corresponding variable in Fig. 1a. This is a *private* variable for the concrete component that is assumed globally to be **AsmNoRW** — i.e. is assumed that other components will neither read nor write it for the entire lifetime of the program. reg_C is used to store the value of the control variable control_C before testing its value with the subsequent **if**-statement.

This refinement of Fig. 1a introduces both new private state (reg_C) as well as changing the level of atomicity for some actions of the original program (the single action of the original program in which control's value was tested

has now been broken into two: one for loading $\mathsf{control}_C$ into reg_C and a second for testing the value of reg_C). Our theory of refinement supports both of these kinds of transformations, including those in which several actions of the abstract component are *optimised* into fewer concrete actions.

2) Refinement Relations: Refinement is phrased formally as a relation between the semantic configurations of the abstract and concrete components. Since the (value-dependent) timing-sensitive noninterference property we consider observes the entirety of memory (as opposed, for instance, to just the input/output events of the program), it requires a fairly strong refinement relation in order to transfer it from the abstract to the concrete component. In particular, the refinement relation must ensure that at each point in the concrete program there exists a corresponding point in the execution of the abstract program. By "corresponding", we require that the values of all variables in the abstract program match the values of their corresponding variables in the concrete program, and that the assumptions (and guarantees – see later in Sec. III) about each abstract variable are identical for their corresponding variable.

This means that the concrete program is free to perform actions not present in the abstract program (such as loading $\mathsf{control}_C$ into reg_C), so long as those actions only modify the newly introduced private variables of the concrete program (here, reg_C). As mentioned above, this doesn't preclude the concrete program from implementing multiple steps of the abstract program in a single step of the concrete program. It does prevent the concrete program from re-ordering actions performed by the abstract program.

These restrictions ensure that the set of reachable states of the concrete program corresponds to that of the abstract program, which is necessary for preserving security. To see why, consider what would happen for instance if the following two statements were reordered: $\mathsf{buffer} := 0$; $\mathsf{control} := 0$. The first ensures that buffer contains non-confidential data, and so is safe to classify as Low (which is the effect of the second statement). Were they reordered, buffer may end up being classified Low while still holding non-Low data — a clear security violation.

3) Dealing with Concurrency from Other Components: In order to allow each component of a concurrent system to be refined independently, our refinement theory must take into account potential effects caused by other concurrently-executing components. Consider the program in Fig. 2a and specifically the statement $\mathsf{x} := \mathsf{y} + \mathsf{z}$ therein. Suppose, as is very common, that the source language semantics models the evaluation of the expression $\mathsf{y} + \mathsf{z}$ as occurring atomically. Suppose, however, that this statement is implemented in the concrete component by multiple statements, the first two of which load y and z into separate registers respectively and the third of which adds them together. Fig. 2b depicts an example of this kind of refinement, expressed in the same tiny imperative language used elsewhere in this paper. In the concrete system, while these multiple instructions are executing, other components may execute in between, possibly modifying the values of (the concrete counterparts of) y and z in memory.

These modifications are problematic for refinement. As far as the source program is concerned, after executing the statement $\mathsf{x} := \mathsf{y} + \mathsf{z}$ it is always true that the sum of the values in the variables y and z is equal to the value in x. However, because of the aforementioned concurrency, the same doesn't follow trivially in the concrete system.

To remedy this problem, we leverage the local assumptions made by each component. Specifically, inspired by [LFF14], we require the refinement relation to be preserved by changes to all variables that the component is assuming may be modified (i.e. all those for which it does not have **AsmNoW** or **AsmNoRW** assumptions). In the example above, this would allow the statement $\mathsf{x} := \mathsf{y} + \mathsf{z}$ to be refined into multiple separate ones as described only if the source component had **AsmNoW** or **AsmNoRW** assumptions for y and z at the time of performing the assignment. This is the reason that the program in Fig. 2a acquires **AsmNoW** for these variables; a similar argument applies to h.

Thus our refinement theory makes explicit the otherwise implicit assumptions made e.g. by the compiler when compiling code that will run in a concurrent context. It does so that components can be soundly refined separately while ensuring that when composed together, the refined concurrent program will not only be secure, but that its reachable states will be a subset of those for the abstract concurrent program.

4) Concrete Coupling Invariants: Knowing that each concrete state corresponds to an abstract one is not sufficient for preserving timing-sensitive noninterference. This is because timing-sensitive noninterference requires the Low-observable timing properties of a program to be independent of non-Low data. When this holds, we term the program *constant time*.

Consider a component like that in Fig. 2a that contains an **if**-statement whose condition examines a non-Low variable. Timing-sensitive noninterference usually requires the **then**- and **else**-branches to take the same number of steps (i.e. the same amount of time, as viewed through the source program's small-step semantics) to execute.[1] The same must be true for the concrete component if noninterference is to be preserved; however, simply requiring that each state of the concrete component corresponds to an abstract state is not enough to ensure this.

To remedy this problem, we introduce the notion of a *concrete coupling invariant*. It relates different executions of the concrete program arising from it having branched on non-Low data, and, by proving that it is maintained, ensures that they each take the same number of steps to execute. For instance, to prove that the refinement in Fig. 2b is security-preserving one uses a coupling invariant that relates the first **skip** statement of the component's **then**-branch to the first statement $\mathsf{reg1}_C := \mathsf{y}_C$ of the component's **else**-branch, relates the second **skip** to $\mathsf{reg2}_C := \mathsf{z}_C$, relates the statement $\mathsf{reg0}_C := \mathsf{y}_C$ to $\mathsf{reg0}_C := \mathsf{reg1}_C + \mathsf{reg2}_C$, and relates the

[1] Fig. 2a satisfies this condition (only) because the addition is evaluated atomically in the language's small-step semantics, as mentioned above.

```
skip   /* h +=_m AsmNoW */;
skip   /* y +=_m AsmNoW */;
skip   /* z +=_m AsmNoW */;
y := 0;
z := 0;
x := y;
if h != 0 then
  x := y
else
  x := y + z
endif
```

(a) A program with a High conditional.

```
skip   /* h_C +=_m AsmNoW */;
skip   /* y_C +=_m AsmNoW */;
skip   /* z_C +=_m AsmNoW */;
y_C := 0;
z_C := 0;
x_C := y_C;
reg3_C := h_C;
if reg3_C != 0 then
  skip;
  skip;
  reg0_C := y_C;
  x_C := reg0_C
else
  reg1_C := y_C;
  reg2_C := z_C;
  reg0_C := reg1_C + reg2_C;
  x_C := reg0_C
endif
```

(b) A refinement of Fig. 2a.

Fig. 2: Compositional refinement of timing-sensitive noninterference.

statement $\mathsf{x}_C := \mathsf{reg0}_C$ to itself. Importantly, coupling invariants need not talk about whether memory contents are secure, but merely whether two possibly different concrete executions are synchronised.

5) Simple Components and Simple Refinements: Components whose execution doesn't branch on non-Low data are trivially constant-time, in that their execution-time (as measured through the number of evaluation steps in the source language's small-step semantics) depends only on Low data. Thus they can be refined securely using only *simple* refinement relations without need for concrete coupling invariants, so long as the refinement relation itself doesn't introduce new branching on non-Low data, or choose to refine programs differently based on the contents of non-Low memory. We derive this result formally as a consequence of our refinement theory in Sec. VI.

III. Compositional, Value-Dependent Noninterference

Before describing our type system (Sec. IV) and refinement theory (Sec. V & Sec. VI), we first briefly describe the compositional theory of value-dependent timing-sensitive noninterference on which they are constructed. We build on [Mur15]'s theory, which is itself a value-dependent extension of a subset of the theory from [MSS11].

1) Preliminaries: The concurrently-executing components share access to a global *memory*, which is modelled as a mapping from a finite set of *variables* to *values*. As mentioned in Sec. II, we restrict our attention to a two-point lattice of security classifications High and Low, where $\mathsf{Low} < \mathsf{High}$. Let $\mathcal{L}_{mem}\ v$ give the (value-dependent) classification of variable v when the memory is *mem*. Let $\mathcal{C}\mathsf{vars}\ v$ denote the (fixed) set of variables that variable v's classification depends on, such that:

$$(\forall x{\in}\mathcal{C}\mathsf{vars}\ y.\ mem_1\ x = mem_2\ x) \longrightarrow \mathcal{L}_{mem_1}\ y = \mathcal{L}_{mem_2}\ y$$

Let $\mathcal{C} \equiv \bigcup_x \mathcal{C}\mathsf{vars}\ x$ denote the set of *control variables*, i.e. those that determine the classification of all other variables. Then we require that these variables are always classified Low in order to prevent changes in variable-classifications from leaking confidential information: $\forall x{\in}\mathcal{C}.\ \mathcal{L}_{mem}\ x = \mathsf{Low} \wedge \mathcal{C}\mathsf{vars}\ x = \emptyset$.

We assume that each component is programmed in a deterministic programming language. (Our type system instantiates this language with the tiny imperative one in which Fig. 1 and Fig. 2 are written.) Let $\langle cmd, mds, mem\rangle$ denote a *local configuration* of an individual component where *cmd* is the currently executing *command*; *mds* is the current *modes state* for that component, which we describe shortly; and *mem* is the current memory. Let $\rightsquigarrow$ denote a transition relation on local configurations that gives the small step operational semantics for the language.

The mode state tracks the current assumptions of an individual component, as well as *guarantees* made by that component. These guarantees are needed to satisfy the assumptions of other components, in order for the proofs of the individual components to compose. Let **AsmNoW**, **AsmNoRW**, **GuarNoW**, and **GuarNoRW** denote *modes* that a component may dynamically associate with each variable. When a component acquires **AsmNoW** on variable v, it assumes no other component will modify v or its classification. Acquiring **AsmNoRW** on v assumes additionally that no variable in $\{v\} \cup \mathcal{C}\mathsf{vars}\ v$ will be read. The modes **GuarNoW** and **GuarNoRW** are the corresponding guarantees for **AsmNoW** and **AsmNoRW**; e.g. acquiring **GuarNoW** on v is a guarantee that the component will not modify v or its classification.

The mode state is a mapping from each mode to the set of variables that currently have that mode. Thus v has e.g. mode **AsmNoW** in mode state *mds* when $v \in mds$ **AsmNoW**.

A *global configuration* models the global state of the system that comprises a collection of concurrently running components. It is a pair: (cms, mem) where *cms* is a list of

command/mode state pairs (cmd_i, mds_i), one for each of the concurrently executing components, and *mem* is the memory (which they all share). $\rightsquigarrow_{\_}$ is the transition relation on global configurations, defined inductively as:

$$\frac{cms_{[i]} = (cmd_i, mds_i) \qquad i < |cms| \qquad \langle cmd_i, mds_i, mem\rangle \rightsquigarrow \langle cmd_i', mds_i', mem'\rangle}{(cms, mem) \rightsquigarrow_i (cms[i := (cmd_i', mds_i')], mem')}$$

For a list *cms*, $cms_{[i]}$ denotes its *i*th element (indexed from 0), and $|cms|$ denotes its length. The expression $cms[i := (cmd_i', mds_i')]$ updates the list *cms* at the *i*th position with (cmd_i', mds_i').

$(cms, mem) \rightsquigarrow_i (cms', mem')$ denotes that the system transitions from global configuration (cms, mem) to configuration (cms', mem') by the *i*th component making an execution step. Such a step updates *i*'s command and mode state pair (cmd_i, mds_i) of *cms*, as well as the global memory *mem*, with the respective values obtained under the transition relation $\rightsquigarrow$ from the local configuration $\langle cms_i, mds_i, mem\rangle$.

Concurrent execution is defined against a fixed *schedule sched*, a finite list prescribing the order in which components are to execute. Execution against *sched* is denoted $\rightarrow_{sched}$, and defined in the natural way.

$c \rightarrow_{[]} c' = (c = c')$

$c \rightarrow_{n \cdot ns} c' = (\exists c''.\ c \rightsquigarrow_n c'' \wedge c'' \rightarrow_{ns} c')$

Here $[]$ is the empty list and $\_ \cdot \_$ the cons operator.

2) Security: We now define the main security properties. There is a *global* system-wide security property, and a *local* security property for each component. These are linked by a central *compositionality* theorem which states that if the local property holds for each component, then the global property holds for the entire system, assuming some side conditions to allow the local properties to compose. It is the local security property that our type system establishes for each component.

a) Global Security: Let $mem_1 =^l mem_2$ denote when the memories mem_1 and mem_2 are Low-equivalent:

$$mem_1 =^l mem_2 \equiv \forall x.\ \mathcal{L}_{mem_1}\ x = \mathsf{Low} \longrightarrow mem_1\ x = mem_2\ x$$

Because all $\mathcal{C}$ variables are Low, it follows straightforwardly that: $mem_1 =^l mem_2 \longrightarrow (\forall x.\ \mathcal{L}_{mem_1}\ x = \mathcal{L}_{mem_2}\ x)$.

Then let sys-secure *cms* be the global security property that denotes when the collection of concurrently executing components *cms* is secure. *cms* is a list of command/initial-mode-state pairs, one for each component.

sys-secure $cms \equiv$
$\forall mem_1\ mem_2.$
  $mem_1 =^l mem_2 \longrightarrow$
  $(\forall sched\ cms_1'\ mem_1'.$
    $(cms, mem_1) \rightarrow_{sched} (cms_1', mem_1') \longrightarrow$
    $(\exists cms_2'\ mem_2'.\ (cms, mem_2) \rightarrow_{sched} (cms_2', mem_2')) \wedge$
    $(\forall cms_2'\ mem_2'.$
      $(cms, mem_2) \rightarrow_{sched} (cms_2', mem_2') \longrightarrow$
      modes-eq $cms_1'\ cms_2' \wedge$
      $(\forall x.\ x \in \mathcal{C} \vee \mathcal{L}_{mem_1'}\ x = \mathsf{Low} \wedge$ readable $cms_1'\ x \longrightarrow$
        $mem_1'\ x = mem_2'\ x)))$

Here modes-eq $cms_1'\ cms_2'$ denotes that the two lists cms_1' and cms_2' agree pointwise on their mode states, and we define readable $cms_1'\ x \equiv \forall (cmd', mds') \in$ set $cms_1'.\ x \notin mds'$ **AsmNoRW** where for a list *xs*, set *xs* denotes the set whose elements are precisely those in *xs*.

sys-secure *cms* asserts that given two initial memories that are Low-equivalent and executing an arbitrary schedule from the first, this execution can always be matched by running the same schedule from the second: in all cases, the two resulting configurations will have the same mode states for each component, and will agree for all control variables (which determine the classification of all others), as well as all Low variables that no component is assuming will not be read — i.e. the two configurations will agree on the values of those variables that must hold Low data.

b) Local Security: The local security property essentially requires showing that each component preserves the following relational property, called Low-equivalent *modulo modes*:

$mem_1 =_{mds}^{\ l} mem_2 \equiv$
$\forall x.\ x \in \mathcal{C} \vee \mathcal{L}_{mem_1}\ x = \mathsf{Low} \wedge x \notin mds$ **AsmNoRW** $\longrightarrow$
  $mem_1\ x = mem_2\ x$

It requires that each component ensures that all $\mathcal{C}$-variables and all Low variables that the component assumes may be read by other components, always contain only Low information. Note that: $mem_1 =_{mds}^{\ l} mem_2 \longrightarrow (\forall x.\ \mathcal{L}_{mem_1}\ x = \mathcal{L}_{mem_2}\ x)$.

To prove that each component maintains this equivalence, the theory requires that a relation $\mathcal{B}$ can be found for each component that relates pairs of executions of the component and ensures that the Low-equivalence modulo modes is always preserved. $\mathcal{B}$ is called a *strong low bisimulation modulo modes* and is defined formally as follows.

$\mathcal{B}$ needs to be *closed under globally consistent changes*, meaning that it is preserved by the actions of the other components in the system, restricted according to the assumptions encoded in the current mode state *mds*. We denote this condition cg-consistent $\mathcal{B}$.

cg-consistent $\mathcal{B} \equiv$
$\forall c_1\ mds\ mem_1\ c_2\ mem_2.$
  $\langle c_1, mds, mem_1\rangle\ \mathcal{B}\ \langle c_2, mds, mem_2\rangle \longrightarrow$
  $(\forall A.\ (\forall x.\ mem_1\ x \neq mem_1\ [||_1\ A]\ x \vee$
      $mem_2\ x \neq mem_2\ [||_2\ A]\ x \longrightarrow$
      writable $mds\ x) \wedge$
    $(\forall x.\ \mathcal{L}_{mem_1}\ x \neq \mathcal{L}_{mem_1\ [||_1\ A]}\ x \longrightarrow$ writable $mds\ x) \wedge$
    $mem_1\ [||_1\ A] =_{mds}^{\ l} mem_2\ [||_2\ A] \longrightarrow$
    $\langle c_1, mds, mem_1\ [||_1\ A]\rangle\ \mathcal{B}\ \langle c_2, mds, mem_2\ [||_2\ A]\rangle)$

cg-consistent $\mathcal{B}$ quantifies over the actions A of other components in the system. An action A models the memory-updates performed by other components and so is a partial mapping from variables to pairs of values (one for each of the memories in the two configurations related by $\mathcal{B}$). We write $mem\ [||_1\ A]$ to denote updating the memory *mem* with the first set of changes in A, and $mem\ [||_2\ A]$ for updating *mem* with the second set. A is restricted to only modify the values or classifications of variables x that are assumed to be writable:

writable $mds\ x \equiv x \notin mds$ **AsmNoW** $\wedge\ x \notin mds$ **AsmNoRW**

Naturally, A is also restricted to preserving Low-equivalence modulo modes.

Let strong-low-bisim-mm $\mathcal{B}$ denote that $\mathcal{B}$ is a strong low bisimulation modulo modes, defined as:

$$\begin{array}{l}\textsf{strong-low-bisim-mm}\ \mathcal{B} \equiv \\ (\textsf{sym}\ \mathcal{B} \wedge \textsf{cg-consistent}\ \mathcal{B}) \wedge \\ (\forall c_1\ mds\ mem_1\ c_2\ mem_2. \\ \quad \langle c_1, mds, mem_1\rangle\ \mathcal{B}\ \langle c_2, mds, mem_2\rangle \longrightarrow \\ \quad mem_1 =_{mds}{}^{l}\ mem_2 \wedge \\ \quad (\forall c_1'\ mds'\ mem_1'. \\ \quad\quad \langle c_1, mds, mem_1\rangle \rightsquigarrow \langle c_1', mds', mem_1'\rangle \longrightarrow \\ \quad\quad (\exists c_2'\ mem_2'. \\ \quad\quad\quad \langle c_2, mds, mem_2\rangle \rightsquigarrow \langle c_2', mds', mem_2'\rangle \wedge \\ \quad\quad\quad \langle c_1', mds', mem_1'\rangle\ \mathcal{B}\ \langle c_2', mds', mem_2'\rangle))))\end{array}$$

$\mathcal{B}$ must be symmetric, closed under globally consistent changes, and imply Low-equivalence modulo modes, as well as being preserved locally by the component.

Finally let com-secure cmd be the local security property that denotes when a single component, whose initial mode state is mds and whose program is cmd, is secure:

$$\begin{array}{l}\textsf{com-secure}\ (cmd, mds) \equiv \\ \forall mem_1\ mem_2. \\ \quad mem_1 =_{mds}{}^{l}\ mem_2 \longrightarrow \\ \quad (\exists \mathcal{B}.\ \textsf{strong-low-bisim-mm}\ \mathcal{B} \wedge \\ \quad\quad \langle cmd, mds, mem_1\rangle\ \mathcal{B}\ \langle cmd, mds, mem_2\rangle)\end{array}$$

3) Compositionality: The compositionality theorem is as follows [Mur15]:

Theorem 3.1:

$$\frac{\forall (cmd, mds) \in \textsf{set}\ cms.\ \textsf{com-secure}\ (cmd, mds) \qquad \forall mem.\ \textsf{sound-mode-use}\ (cms, mem)}{\textsf{sys-secure}\ cms}$$

For the local security properties to compose, this theorem requires that each component always meets the assumptions of all others: $\forall mem.$ sound-mode-use (cms, mem).

sound-mode-use essentially requires that each component (1) guarantees to meet the assumptions of all others and (2) always adheres to its own guarantees. (1) requires that whenever a component has an **AsmNoRW** (respectively **AsmNoW**) assumption for a variable v, that all other components have **GuarNoRW** (resp. **GuarNoW**) for v. (2) requires that whenever a component has the **GuarNoRW** guarantee for variable v, then its next step of execution doesn't depend on nor modify any variable in $\{v\} \cup \mathcal{C}\textsf{vars}\ v$; and whenever it has **GuarNoW** for v then it doesn't alter v's value nor its classification.

Recent work has shown how to verify that these conditions are enforced [MMOPW15], or how to enforce them using dynamic monitoring [ACM15].

IV. Flow-Sensitive Dependent Type System

We now present our value-dependent type system, whose purpose is to prove components locally secure. Its soundness proof, mechanised in Isabelle/HOL, constructs an appropriate bisimulation $\mathcal{B}$, and proves that it is a strong low bisimulation modulo modes. Its structure is similar to the Isabelle/HOL soundness proof of [MSS11]'s non-value-dependent type system; however ours is considerably more involved (at almost triple the size).

The type system is defined over the simple imperative language of [MSS11], extended with the illustrative synchronisation primitive **await** [PN02]. The statement **await** e **do** c **done** blocks execution until the condition e is satisfied, and then *atomically* executes the body c.

A. Deeply Embedded Types

Recall from Sec. II that value-dependent security classifications (i.e. types) in our type system are deeply embedded as sets of predicates t. For a set of predicates P, we write $\lfloor P \rfloor_{mem}$ when all predicates in P hold in memory mem. Then let $[\![t]\!]_{mem}$ denote the *interpretation* of type t in memory mem, defined as:

$$[\![t]\!]_{mem} \equiv \textit{if}\ \lfloor t \rfloor_{mem}\ \textit{then}\ \textsf{Low}\ \textit{else}\ \textsf{High}$$

Types are interpreted as Low when all of their predicates hold, and as High otherwise.

We say that one type t is a *subtype* of another t' under predicates P, when, in each memory that satisfies P, t's interpretation is always $\leq$ to t''s. We can phrase this property conveniently in terms of *entailment* between sets of predicates. Let $P \vdash P'$ denote when the set of predicates P entails the set of predicates P', namely when $\forall mem.\ \lfloor P \rfloor_{mem} \longrightarrow \lfloor P' \rfloor_{mem}$. Then we write $t \leq:_P t'$ to denote that t is a subtype of t' under predicates P:

$$t \leq:_P t' \equiv (P \cup t') \vdash t$$

Then: $t \leq:_P t' = (\forall mem.\ \lfloor P \rfloor_{mem} \longrightarrow [\![t]\!]_{mem} \leq [\![t']\!]_{mem})$ as required. The characterisation of subtyping in terms of predicate entailment makes automating subtyping judgements more straightforward, which assists the automatic application of our typing rules for proving components secure.

Type *equivalence* is used when rewriting typing contexts, as mentioned earlier in Sec. II-A. It is simply subtyping in both directions:

$$t =:_P t' \equiv t \leq:_P t' \wedge t' \leq:_P t$$

We say that a type is *wellformed* when its predicates refer only to control variables (i.e. those in $\mathcal{C}$ — see Sec. III). Recall that the value-dependent classification of variable v in memory mem is given by function $\mathcal{L}_{mem}\ v$. Then let $\mathcal{L}\textsf{type}\ v$ denote a set of predicates such that $\mathcal{L}_{mem}\ x = [\![\mathcal{L}\textsf{type}\ x]\!]_{mem}$ and $\mathcal{C}\textsf{vars}\ x = \textsf{vars-of-type}\ (\mathcal{L}\textsf{type}\ x)$, where vars-of-type t denotes the set of variables appearing in predicates in t. $\mathcal{L}\textsf{type}\ v$ encodes v's value-dependent classification as a dependent type in our type system. All types given by $\mathcal{L}\textsf{type}$ are clearly wellformed.

B. Contexts and Wellformedness

Recall that our type system tracks three contexts: (1) Γ, the flow-sensitive typing environment; (2) $\mathcal{S}$, which tracks which variables are currently stable (i.e. the two sets of variables that

have **AsmNoW** and **AsmNoRW** respectively); and (3) P, the set of predicates known to hold currently.

P tracks only those predicates known to hold currently, in order to keep its size under control during type checking. However, as mentioned in Sec. II-A, this restricts the variables and types in Γ, and the predicates in P to just those that are stable, meaning that they refer only to stable variables (i.e. those in $\mathcal{S}$).

Our typing rules ensure that Γ tracks only stable, non-$\mathcal{C}$ variables. We say that the three contexts Γ, $\mathcal{S}$ and P are *wellformed* with respect to the mode state *mds* when: (1) all types in Γ are wellformed (as defined above) and stable, (2) Γ's domain is all stable non-$\mathcal{C}$ variables, (3) $\mathcal{S}$ agrees with *mds*, i.e. $\mathcal{S} = (mds\ \textbf{AsmNoW}, mds\ \textbf{AsmNoRW})$, and (4) all predicates in P are stable. We denote this condition $\mathsf{wf}\ mds\ \Gamma\ \mathcal{S}\ P$.

We lift the notion of type equivalence under a set of predicates P to typing contexts, overloading the notation.

$$\Gamma =:_P \Gamma' \equiv$$
$$\mathsf{dom}\ \Gamma = \mathsf{dom}\ \Gamma' \wedge (\forall x{\in}\mathsf{dom}\ \Gamma'.\ \lfloor \Gamma\ x \rfloor =:_P \lfloor \Gamma'\ x \rfloor)$$

Here, $\mathsf{dom}\ \Gamma$ denotes the domain of Γ and $\lfloor \Gamma\ x \rfloor$ denotes the type that Γ has recorded for any $x \in \mathsf{dom}\ \Gamma$.

Finally, we lift Γ from a partial- to a total-function on variables x, denoted $\Gamma\langle x \rangle$, defined as:

$$\Gamma\langle x \rangle \equiv \mathit{if}\ x \in \mathsf{dom}\ \Gamma\ \mathit{then}\ \lfloor \Gamma\ x \rfloor\ \mathit{else}\ \mathcal{L}\mathsf{type}\ x$$

C. Typing Rules

The typing rule for expressions e takes the upper bound of the types of the e's variables, denoted $\mathsf{vars\text{-}of}\ e$.

$$\Gamma \vdash e : \bigcup_{x \in \mathsf{vars\text{-}of}\ e} \Gamma\langle x \rangle$$

The typing rules for commands are shown in Fig. 3. SEQ and SKIP mirror standard typing rules for these constructs. As elsewhere, WHILE requires the loop-guard e to be Low in order to satisfy timing-sensitive noninterference. It does so by asserting that e's type t be Low (i.e. all predicates in t hold), under the current predicates P: $P \vdash t$. When typing the loop-body c, the notation $P +_{\mathcal{S}} e$ adds the loop condition e to P so long as all of the variables it mentions are stable in $\mathcal{S}$. This allows reasoning under the loop guard as in Hoare logic.

The AWAIT rule naturally requires the condition e to be Low: since the component will block until e is satisfied, its timing behaviour necessarily depends on e's classification. As with the WHILE rule, when typing the body c, e is added to the predicate set P provided it is stable in $\mathcal{S}$.

The IF rule also requires the condition e to be Low. While we could extend our type system to accommodate non-Low conditionals for **if**-statements (e.g. by following the approach of [MSS11]), doing so is largely orthogonal to the issues of value-dependent noninterference, which is the focus of this paper. When typing the **then** (c_1) and **else** (c_2) branches respectively, the loop condition (respectively its negation) is added to the predicate set P so long as it is stable. The IF rule is designed to be as general as possible. Thus, when typing the **then** and **else** branches, it allows for the fact that the post-contexts may differ between the two branches. Specifically, the typing context and predicate sets may differ. The mode state must be identical to prevent trivial information leaks. To handle this difference, the rule allows the differing post contexts to be unified into a common typing environment Γ''' and predicate set P'''.

If used in this most general form, the IF rule requires the user to establish that the unified post-contexts are guaranteed to be wellformed. However, given a particular expression language, the process of finding the unified contexts can be automated along with these wellformedness obligations. We have implemented specialised forms of the IF rule in our Isabelle/HOL formalisation that both infer the post-contexts and safely discard the wellformedness obligations because the inferred contexts are wellformed by construction. Fig. 4 depicts an example specialised rule for an expression language that includes boolean implication.

Our type system includes three rules for assignments $x := e$, depending on the variable x being assigned to: AS1 for non-$\mathcal{C}$ variables not in Γ, AS2 for (necessarily non-$\mathcal{C}$) variables in Γ, and AS$\mathcal{C}$ for $\mathcal{C}$-variables (necessarily not in Γ). Each of these three rules updates the set of predicates P by making use of a user-supplied postcondition transformer post that, given a set of predicates P and an assignment $x := e$, calculates a new set of predicates guaranteed to hold after the assignment to x is performed, under the assumption that all predicates in P held beforehand. The new set of predicates must be restricted to those that are stable under $\mathcal{S}$, which we denote $\restriction \mathcal{S}$. This allows the type system to propagate information from past assignment statements. We assume that post is always sound:

$$\frac{\langle e, mem \rangle \downarrow v}{\lfloor P \rfloor_{mem} \longrightarrow \lfloor \mathsf{post}\ P\ \langle x := e \rangle \rfloor_{mem(x := v)}}$$

Here $\langle e, mem \rangle \downarrow v$ denotes that expression e evaluates to value v in memory mem, while $mem(x := v)$ denotes memory mem with the value of variable x updated to v.

AS1 requires the type of the expression e to not exceed x's type. AS2 requires e's type t to be stable, in that it doesn't mention any unstable variables. Additionally, if x is not assumed to be not read by other components (i.e. $x \notin \mathsf{snd}\ \mathcal{S}$ and so doesn't have **AsmNoRW**), then t cannot exceed x's type after the assignment is performed. Thus this rule allows storing High data in Low variables that are **AsmNoRW**.

AS$\mathcal{C}$ is the most interesting assignment rule, as it has to account for potential changes in classification of *other* variables, caused by changing x when $x \in \mathcal{C}$. It naturally requires that the expression e is Low. But in addition, it requires that Γ will not be invalidated by the assignment, by requiring that no types in Γ mention x. As with similar restrictions, context-rewriting can always be used to work around this one, via the REWRITE rule that we explain shortly. To account for when x may change another variable v's classification, AS$\mathcal{C}$ requires that any such v that is not **AsmNoRW** contains data at the time of the assignment that (1) is Low ($P \vdash \Gamma\langle v \rangle$), and (2) does not exceed its (new) classification after the assignment ($\Gamma\langle v \rangle \leq:_{P'} \mathcal{L}\mathsf{type}\ v$).

$$\frac{\vdash \Gamma,\mathcal{S},P\ \{c_1\}\ \Gamma',\mathcal{S}',P' \qquad \vdash \Gamma',\mathcal{S}',P'\ \{c_2\}\ \Gamma'',\mathcal{S}'',P''}{\vdash \Gamma,\mathcal{S},P\ \{(c_1\ ;\ c_2)\}\ \Gamma'',\mathcal{S}'',P''}\ \textsc{Seq} \qquad \frac{x \notin \mathsf{dom}\ \Gamma \cup \mathcal{C} \qquad \Gamma \vdash e : t \qquad t \leq:_P \mathcal{L}\mathsf{type}\ x}{\vdash \Gamma,\mathcal{S},P\ \{x := e\}\ \Gamma,\mathcal{S},\mathsf{post}\ P\ \langle x := e\rangle \restriction \mathcal{S}}\ \textsc{As1}$$

$$\frac{}{\vdash \Gamma,\mathcal{S},P\ \{\textbf{skip}\}\ \Gamma,\mathcal{S},P}\ \textsc{Skip} \qquad \frac{\Gamma \vdash e : t \qquad P \vdash t \qquad \vdash \Gamma,\mathcal{S},P +_{\mathcal{S}} e\ \{c\}\ \Gamma,\mathcal{S},P}{\vdash \Gamma,\mathcal{S},P\ \{\textbf{while}\ e\ \textbf{do}\ c\ \textbf{done}\}\ \Gamma,\mathcal{S},P}\ \textsc{While}$$

$$\frac{\Gamma \vdash e : t \qquad P \vdash t \qquad \vdash \Gamma,\mathcal{S},P +_{\mathcal{S}} e\ \{c\}\ \Gamma',\mathcal{S}',P'}{\vdash \Gamma,\mathcal{S},P\ \{\textbf{await}\ e\ \textbf{do}\ c\ \textbf{done}\}\ \Gamma',\mathcal{S}',P'}\ \textsc{Await}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : t \qquad P \vdash t \\ \vdash \Gamma,\mathcal{S},P +_{\mathcal{S}} e\ \{c_1\}\ \Gamma',\mathcal{S}',P' \qquad \vdash \Gamma,\mathcal{S},P +_{\mathcal{S}} \neg\ e\ \{c_2\}\ \Gamma'',\mathcal{S}',P'' \qquad \Gamma' =:_{P'} \Gamma''' \qquad \Gamma'' =:_{P''} \Gamma''' \qquad P' \vdash P''' \\ P'' \vdash P''' \qquad \forall mds.\ \mathsf{wf}\ mds\ \Gamma'\ \mathcal{S}'\ P' \longrightarrow \mathsf{wf}\ mds\ \Gamma'''\ \mathcal{S}'\ P''' \qquad \forall mds.\ \mathsf{wf}\ mds\ \Gamma''\ \mathcal{S}'\ P'' \longrightarrow \mathsf{wf}\ mds\ \Gamma'''\ \mathcal{S}'\ P'''\end{array}}{\vdash \Gamma,\mathcal{S},P\ \{\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{endif}\}\ \Gamma''',\mathcal{S}',P'''}\ \textsc{If}$$

$$\frac{x \in \mathsf{dom}\ \Gamma \qquad \Gamma \vdash e : t \qquad \mathsf{stable}\ \mathcal{S}\ t \qquad P' = \mathsf{post}\ P\ \langle x := e\rangle \restriction \mathcal{S} \qquad x \notin \mathsf{snd}\ \mathcal{S} \longrightarrow t \leq:_{P'} \mathcal{L}\mathsf{type}\ x}{\vdash \Gamma,\mathcal{S},P\ \{x := e\}\ \Gamma(x \mapsto t),\mathcal{S},P'}\ \textsc{As2}$$

$$\frac{\begin{array}{c}x \in \mathcal{C} \qquad \Gamma \vdash e : t \qquad P \vdash t \qquad \forall v{\in}\mathsf{dom}\ \Gamma.\ x \notin \mathsf{vars\text{-}of\text{-}type}\ \lfloor \Gamma\ v \rfloor \\ P' = \mathsf{post}\ P\ \langle x := e\rangle \restriction \mathcal{S} \qquad \forall v.\ x \in \mathcal{C}\mathsf{vars}\ v \wedge v \notin \mathsf{snd}\ \mathcal{S} \longrightarrow P \vdash \Gamma\langle v\rangle \wedge \Gamma\langle v\rangle \leq:_{P'} \mathcal{L}\mathsf{type}\ v\end{array}}{\vdash \Gamma,\mathcal{S},P\ \{x := e\}\ \Gamma,\mathcal{S},P'}\ \textsc{As}\mathcal{C}$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma \oplus_{\mathcal{S}} upd \qquad \mathcal{S}' = \mathcal{S} \oplus upd \qquad P' = P \restriction \mathcal{S}' \\ \vdash \Gamma',\mathcal{S}',P'\ \{c\}\ \Gamma'',\mathcal{S}'',P'' \qquad \forall x.\ \Gamma\langle x\rangle \leq:_{P'} \Gamma'\langle x\rangle \qquad \mathsf{anno\text{-}stable}\ \Gamma\ \mathcal{S}\ upd \qquad \mathsf{anno\text{-}sec}\ \Gamma\ \mathcal{S}\ P\ upd\end{array}}{\vdash \Gamma,\mathcal{S},P\ \{c \quad /*\ upd\ */\}\ \Gamma'',\mathcal{S}'',P''}\ \textsc{Anno}$$

$$\frac{\begin{array}{c}\vdash \Gamma_1,\mathcal{S},P_1\ \{c\}\ \Gamma_1{}',\mathcal{S}',P_1{}' \qquad \Gamma_2 =:_{P_2} \Gamma_1 \qquad \forall mds.\ \mathsf{wf}\ mds\ \Gamma_2\ \mathcal{S}\ P_2 \longrightarrow \mathsf{wf}\ mds\ \Gamma_1\ \mathcal{S}\ P_1 \\ \forall mds.\ \mathsf{wf}\ mds\ \Gamma_1{}'\ \mathcal{S}'\ P_1{}' \longrightarrow \mathsf{wf}\ mds\ \Gamma_2{}'\ \mathcal{S}'\ P_2{}' \qquad \Gamma_1{}' =:_{P_1{}'} \Gamma_2{}' \qquad P_2 \vdash P_1 \qquad P_1{}' \vdash P_2{}'\end{array}}{\vdash \Gamma_2,\mathcal{S},P_2\ \{c\}\ \Gamma_2{}',\mathcal{S}',P_2{}'}\ \textsc{Rewrite}$$

Fig. 3: Typing rules.

$$\frac{\begin{array}{c}\Gamma \vdash e : t \qquad P \vdash t \qquad \Gamma,\mathcal{S},P +_{\mathcal{S}} e\ \{c_1\}\ \Gamma',\mathcal{S}',P' \qquad \Gamma,\mathcal{S},P +_{\mathcal{S}} \neg\ e\ \{c_2\}\ \Gamma',\mathcal{S}',P'' \\ P' \vdash \{e\} \qquad P'' \vdash \{\neg\ e\} \qquad P''' = ((\lambda x.\ e \longrightarrow x)\ `\ P' \cup (\lambda x.\ \neg\ e \longrightarrow x)\ `\ P'') \restriction \mathcal{S}'\end{array}}{\Gamma,\mathcal{S},P\ \{\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{endif}\}\ \Gamma',\mathcal{S}',P'''}$$

Fig. 4: A specialised version of the IF rule that automatically infers wellformed post-contexts.

The ANNO rule parallels that of [MSS11]'s non-value-dependent type system, except that is has the side conditions anno-stable and anno-sec described shortly. Here *upd* is an annotation like v $+\!\!=_m$ **AsmNoW** or v $-\!\!=_m$ **GuarNoRW**. The notation $\Gamma \oplus_{\mathcal{S}} upd$ updates Γ based on the annotation *upd* and $\mathcal{S}$: any variable made newly stable by *upd* is added to Γ (by mapping it to its $\mathcal{L}$type); likewise any variable made unstable by *upd* given $\mathcal{S}$ is removed from Γ. The notation $\mathcal{S} \oplus upd$ updates $\mathcal{S}$ based on *upd* in the obvious way. anno-stable checks that releasing an assumption on a control variable v won't cause any types in Γ to become unstable (i.e. no types in Γ mention v), and that acquiring an assumption on a new non-control variable v that causes it to be added to Γ won't lead to Γ containing an unstable type (i.e. all of v's control variables are already stable). anno-sec checks that if one releases a **AsmNoRW** assumption on a variable v for which one also has the **AsmNoW** assumption (making the variable readable by others), then the classification of the variable's data $\lfloor \Gamma\ v \rfloor$ does not exceed its $\mathcal{L}$type. This condition is needed in addition to the check $\forall x.\ \Gamma\langle x\rangle \leq:_{P'} \Gamma'\langle x\rangle$ (from [MSS11]) because in our type system Γ tracks all stable variables.

Finally, the REWRITE rule allows typing contexts and predicate sets to be rewritten, by replacing types with equivalent ones (under the appropriate predicate set), and by weakening and strengthening the pre- and post-predicate sets respectively, as in Hoare logic. As mentioned earlier, we include this rule to deal with the restriction that all types and predicates must always be stable, and the type system has been designed to facilitate the automatic application of this rule.

D. Soundness

Let Γ_0 and mds_0 denote the empty typing context and mode state respectively. Then:

Theorem 4.1:

$$\frac{\vdash \Gamma_0,(\emptyset,\ \emptyset),\emptyset\ \{cmd\}\ \Gamma',\mathcal{S}',P'}{\mathsf{com\text{-}secure}\ (cmd,\ \mathsf{mds}_0)}$$

Our Isabelle/HOL formalisation contains the (slightly more detailed) statement for an arbitrary initial mode state.

V. Compositional Refinement

We now present a notion of refinement for each component of a concurrent system. If a component is com-secure, then it will remain so when refined in this way. Additionally this notion of refinement composes and can be used to preserve the side conditions of Theorem 3.1. Thus, given a system proved secure using Theorem 3.1, each of whose components has been refined in this way, we can prove the concrete system secure by applying Theorem 3.1 to it.

We first lay the groundwork by defining a number of technical conditions required by our notion of refinement.

A. Memory-and-Mode-Preserving Refinements

We restrict our attention in this paper to concrete refinements that may have extra variables not present in the abstract program, but in which each abstract variable is represented by a unique corresponding concrete variable. Thus let $\mathsf{var}_C\mathsf{of}$ be an injective function from abstract to concrete variables: for an abstract variable v, $\mathsf{var}_C\mathsf{of}\ v$ denotes its concrete counterpart. For the example refinement of Fig. 1a into Fig. 1b, $\mathsf{var}_C\mathsf{of}\ \mathsf{buffer} = \mathsf{buffer}_C$, but $\forall v.\ \mathsf{var}_C\mathsf{of}\ v \neq \mathsf{reg}_C$.

Recall from Sec. III that the semantics of components is given by a small-step relation on configurations $\langle cms, mds, mem\rangle$. Formally, a *refinement relation* $\mathcal{R}$ relates configurations $\langle cms_A, mds_A, mem_A\rangle_A$ of an abstract component to corresponding configurations $\langle cms_C, mds_C, mem_C\rangle_C$ of its concrete refinement. We restrict our attention here to refinement relations $\mathcal{R}$ that preserve the abstract contents of memory mem_A, as well as the abstract mode state mds_A which, recall, tracks the assumptions and guarantees of the component as it executes. This helps us transfer the abstract component's security proof to its concrete refinement. Thus let preserves-modes-mem $\mathcal{R}$ denote when the refinement relation $\mathcal{R}$ *preserves modes and memory*:

$$\begin{aligned}&\mathsf{preserves\text{-}modes\text{-}mem}\ \mathcal{R} \equiv\\ &\forall\, c_A\ mds_A\ mem_A\ c_C\ mds_C\ mem_C.\\ &\quad \langle c_A, mds_A, mem_A\rangle_A\ \mathcal{R}\ \langle c_C, mds_C, mem_C\rangle_C \longrightarrow\\ &\quad (\forall x_A.\ mem_A\ x_A = mem_C\ (\mathsf{var}_C\mathsf{of}\ x_A)) \wedge\\ &\quad (\forall m.\ \mathsf{var}_C\mathsf{of}\ `\ mds_A\ m = \mathsf{range}\ \mathsf{var}_C\mathsf{of} \cap mds_C\ m)\end{aligned}$$

Here, $f\ `\ A$ denotes the image of A under f, i.e. the set obtained by applying the function f to every element of the set A; while $\mathsf{range}\ \mathsf{var}_C\mathsf{of}$ is the set of concrete variables that correspond to abstract variables. Recall that the mode state, e.g. mds_A is a mapping from modes m (e.g. **AsmNoW** etc.) to the set of variables with that mode.

We define functions $\mathsf{mem}_A\mathsf{of}$ and $\mathsf{mds}_A\mathsf{of}$ for mapping concrete memories mem_C and mode states mds_C to their corresponding abstract counterparts.

$$\mathsf{mem}_A\mathsf{of}\ mem_C \equiv \lambda x_A.\ mem_C\ (\mathsf{var}_C\mathsf{of}\ x_A)$$

$$\mathsf{mds}_A\mathsf{of}\ mds_C \equiv \lambda m.\ \mathsf{inv}\ \mathsf{var}_C\mathsf{of}\ `\ (\mathsf{range}\ \mathsf{var}_C\mathsf{of} \cap mds_C\ m)$$

Here, $\mathsf{inv}\ f$ denotes function f's inverse. Then it follows that

$$\begin{aligned}&\mathsf{preserves\text{-}modes\text{-}mem}\ \mathcal{R} \wedge\\ &\langle c_A, mds_A, mem_A\rangle_A\ \mathcal{R}\ \langle c_C, mds_C, mem_C\rangle_C \longrightarrow\\ &mem_A = \mathsf{mem}_A\mathsf{of}\ mem_C \wedge mds_A = \mathsf{mds}_A\mathsf{of}\ mds_C\end{aligned}$$

To ensure that corresponding variables have their classifications interpreted the same way in the abstract and concrete programs, we assume a number of obvious compatibility conditions on the value-dependent classification functions of both. Let $\mathcal{L}_{mem_A}\ x_A$ denote the classification of abstract variable x_A in abstract memory mem_A, and $\mathcal{L}_{mem_C}\ x_C$ likewise for the concrete refinement; $\mathcal{C}\mathsf{vars}_A\ x_A$ denote the (abstract) control variables of abstract variable x_A, and $\mathcal{C}\mathsf{vars}_C\ x_C$ likewise for the concrete refinement; and $\mathcal{C}_A$ and $\mathcal{C}_C$ denote respectively the set of abstract and concrete control variables. Then we assume:

$$\mathcal{L}_{\mathsf{mem}_A\mathsf{of}\ mem_C}\ x_A = \mathcal{L}_{mem_C}\ \mathsf{var}_C\mathsf{of}\ x_A$$

$$\mathsf{var}_C\mathsf{of}\ `\ \mathcal{C}\mathsf{vars}_A\ x_A = \mathcal{C}\mathsf{vars}_C\ (\mathsf{var}_C\mathsf{of}\ x_A)$$

$$\mathcal{C}_C = \mathsf{var}_C\mathsf{of}\ `\ \mathcal{C}_A$$

B. Concurrency-Aware Refinement

Recall from Sec. II-B that, in order to take into account the possible effects caused by concurrently running components while still allowing each component to be refined in isolation, we require that each component's refinement relation be preserved by any changes that can occur under the component's current assumptions. This idea is inspired directly by condition (5) of the RGSim framework [LFF14] and can also be seen as an analogue of the cg-consistent property (see Sec. III) for the refinement relation. It is the key ingredient that makes the refinement theory compositional.

We denote this requirement closed-others:

$$\begin{aligned}&\mathsf{closed\text{-}others}\ \mathcal{R} \equiv\\ &\forall\, c_A\ c_C\ mds_C\ mem_C\ mem_C'.\\ &\quad \langle c_A, \mathsf{mds}_A\mathsf{of}\ mds_C, \mathsf{mem}_A\mathsf{of}\ mem_C\rangle_A\ \mathcal{R}\ \langle c_C, mds_C, mem_C\rangle_C\\ &\wedge\\ &\quad (\forall x.\ mem_C\ x \neq mem_C'\ x \longrightarrow \mathsf{writable}\ mds_C\ x) \wedge\\ &\quad (\forall x.\ \mathcal{L}_{mem_C}\ x \neq \mathcal{L}_{mem_C'}\ x \longrightarrow \mathsf{writable}\ mds_C\ x) \longrightarrow\\ &\quad \langle c_A, \mathsf{mds}_A\mathsf{of}\ mds_C, \mathsf{mem}_A\mathsf{of}\ mem_C'\rangle_A\ \mathcal{R}\ \langle c_C, mds_C, mem_C'\rangle_C\end{aligned}$$

C. Private Variables

Our theory of refinement supports the introduction of new variables (memory locations) in the concrete program. This allows, for instance, refining a programming language source semantics that doesn't explicitly model the runtime stack in memory, to one that does. Or to introduce a concrete heap from which dynamic allocation is performed, and so on.

In order to ensure that security is preserved from the abstract to the concrete system, however, we require that any new memory that is modified, or has its classification decreased,

by a component is also private to that component. We do so by stipulating that the refinement relation $\mathcal{R}$ allows such changes to occur only to variables assumed **AsmNoRW**, and that once any such variable is assumed private in this way, it stays so thereafter. We denote formally this requirement new-vars-private $\mathcal{R}$:

new-vars-private $\mathcal{R} \equiv$
$\forall\, c_A\ mds_A\ mem_A\ c_C\ mds_C\ mem_C.$
$\quad \langle c_A, mds_A, mem_A\rangle_A\ \mathcal{R}\ \langle c_C, mds_C, mem_C\rangle_C \longrightarrow$
$\quad (\forall\, c_C{}'\ mds_C{}'\ mem_C{}'.$
$\qquad \langle c_C, mds_C, mem_C\rangle_C \rightsquigarrow_C \langle c_C{}', mds_C{}', mem_C{}'\rangle_C \longrightarrow$
$\qquad (\forall\, v_C.\ (mem_C{}'\ v_C \neq mem_C\ v_C \vee$
$\qquad\quad \mathcal{L}_{mem_C{}'}\ v_C < \mathcal{L}_{mem_C}\ v_C) \wedge$
$\qquad\quad v_C \notin$ range var$_C$of $\longrightarrow$
$\qquad\quad v_C \in mds_C{}'$ **AsmNoRW**) $\wedge$
$\qquad mds_C$ **AsmNoRW** $-$ range var$_C$of $\subseteq mds_C{}'$ **AsmNoRW** $-$ range var$_C$of)

Here, $\rightsquigarrow_C$ denotes the small-step semantics of the refined component. Note that this requirement excludes those refinements that introduce new variables used for communication or interaction between components.

D. Preserving Security

Now, finally, we present the main requirement of the refinement relation for preserving com-secure, over the aforementioned technical ones. As mentioned earlier, it rests on the notion of a concrete coupling invariant for transferring, from the abstract component to its concrete refinement, the fact that the component's Low-observable timing properties are independent of non-Low data. If such an invariant can be found, it proves that the concrete component's security can be derived from the abstract one's.

To transfer security from the abstract component to its concrete refinement, we define a function $\mathcal{B}_C$of for constructing a strong low bisimulation modulo modes for the concrete component, from the one $\mathcal{B}$ used to prove the abstract component's security. This function takes the refinement relation $\mathcal{R}$ and the coupling invariant $\mathcal{I}$ as arguments.

$\mathcal{B}_C$of $\mathcal{B}\ \mathcal{R}\ \mathcal{I} \equiv$
$\{(\langle c_{1C}, mds_C, mem_{1C}\rangle_C,$
$\quad \langle c_{2C}, mds_C, mem_{2C}\rangle_C) \mid \exists\, lc_{1A}\ lc_{2A}.$
$lc_{1A}\ \mathcal{R}\ \langle c_{1C}, mds_C, mem_{1C}\rangle_C \wedge$
$lc_{2A}\ \mathcal{R}\ \langle c_{2C}, mds_C, mem_{2C}\rangle_C \wedge$
$lc_{1A}\ \mathcal{B}\ lc_{2A} \wedge$
$mem_{1C} =_{mds_C}{}^{l}\ mem_{2C} \wedge$
$\langle c_{1C}, mds_C, mem_{1C}\rangle_C\ \mathcal{I}\ \langle c_{2C}, mds_C, mem_{2C}\rangle_C\}$

$\mathcal{B}_C$of constructs a candidate concrete bisimulation that relates any two configurations $\langle c_{1C}, mds_C, mem_{1C}\rangle_C$ and $\langle c_{2C}, mds_C, mem_{2C}\rangle_C$ when each has a corresponding (under $\mathcal{R}$) abstract configuration lc_{1A} and lc_{2A} respectively, such that the two abstract configurations are related by the abstract bisimulation $\mathcal{B}$. Additionally, the two concrete configurations must also be related by the coupling invariant $\mathcal{I}$ and be Low-equivalent modulo modes which, recall, requires that the values of all $\mathcal{C}$-variables, plus all Low-variables that are assumed readable, must agree between the two.

To show that any such $\mathcal{B}_C$of $\mathcal{B}\ \mathcal{R}\ \mathcal{I}$ really is a strong low bisimulation modulo modes, it suffices to show that the coupling invariant $\mathcal{I}$ is *preserved with the refinement relation*. We denote this property coupling-inv-pres $\mathcal{B}\ \mathcal{R}\ \mathcal{I}$. Its formal definition appears in Fig. 5a and is also depicted graphically in Fig. 5b, to which we will refer.

Fig. 5b depicts a cube whose vertices are labelled $1C$, $2C'$, $1A$ etc. Each corresponds to a configuration in the definition of coupling-inv-pres whose c and *mem* subscripts are the vertex's label. Call the plane on which the vertices $1C$, $1A$, $1A'$ and $1C'$ lie the *front* of the cube, and the plane on which the other four vertices lie the cube's *back*. Other sides of the cube can be named unambiguously.

The line labelled $\mathcal{R}$ that connects $1A$ and $1C$ indicates that the corresponding configurations are related by $\mathcal{R}$. Notice that this is also the first premise of coupling-inv-pres. Solid lines indicate premises and dotted lines indicate conclusions.

Focusing just on the front side of the cube for now, this side depicts a fairly typical definition of refinement (that preserves abstract memory contents), although one that on its own does not preserve timing-sensitive security. It states that each single execution step ($1C$ to $1C'$) of the concrete component must be able to be matched by the abstract component, beginning in an $\mathcal{R}$-related configuration ($1A$), by performing some number n of steps ($1A$ to $1A'$), so that the resulting configurations ($1A'$ and $1C'$) are $\mathcal{R}$-related.

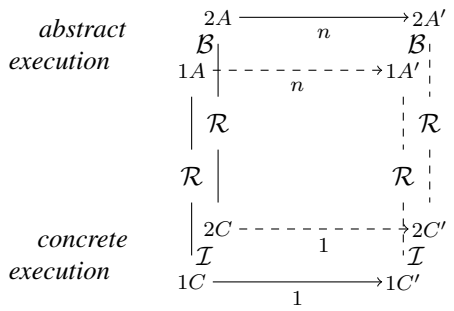
The remainder of the diagram accounts for the preservation of timing-sensitive security. It asks to consider any abstract configuration $2A$ that is $\mathcal{B}$-related to $1A$, any abstract configuration $2A'$ reached from $2A$ by performing n steps, and any concrete configuration $2C$ that is $\mathcal{R}$-related to $2A$ and $\mathcal{I}$-related to $1C$. Then to preserve security, there must exist a concrete configuration $2C'$ reached in a single step from $2C$ that is $\mathcal{R}$-related to $2A'$, and the coupling invariant $\mathcal{I}$ must relate $1C'$ and $2C'$.

Notice that the back plane of the figure *inverts* the ordinary direction of refinement (from the front plane): it requires finding concrete transitions to match abstract ones, whereas the front plane requires finding abstract transitions to match concrete ones. It is well known that one can reverse the direction of refinement for deterministic programs. One way to view the obligations imposed by the back plane of the figure then is as a way of proving that the concrete component is in some sense deterministic. Because it forces finding related configurations after the same number *n* of steps as for the front plane, it effectively requires proving that the Low-observable timing behaviour of the concrete component is deterministic for any pair of concrete Low-equivalent concrete configurations. Thus, the back plane can be seen as a way to prove that the concrete component's Low-observable timing properties are independent of non-Low data.

We can now prove the main theorem that justifies when $\mathcal{B}_C$of $\mathcal{B}\ \mathcal{R}\ \mathcal{I}$ is a strong low bisimulation modulo modes. We collect the conditions discussed so far into a single definition of what it means for a refinement relation $\mathcal{R}$ to preserve security, proved by an abstract bisimulation $\mathcal{B}$, with

coupling-inv-pres $\mathcal{B}\ \mathcal{R}\ \mathcal{I} \equiv$
$\forall\, c_{1A}\ mds_A\ mem_{1A}\ c_{1C}\ mds_C\ mem_{1C}.$
$\langle c_{1A}, mds_A, mem_{1A}\rangle_A\ \mathcal{R}\ \langle c_{1C}, mds_C, mem_{1C}\rangle_C \longrightarrow$
$(\forall\, c_{1C}'\ mds_C'\ mem_{1C}'.$
$\langle c_{1C}, mds_C, mem_{1C}\rangle_C \rightsquigarrow_C \langle c_{1C}', mds_C', mem_{1C}'\rangle_C \longrightarrow$
$(\exists n\ c_{1A}'\ mds_A'\ mem_{1A}'.$
$\langle c_{1A}, mds_A, mem_{1A}\rangle_A \rightsquigarrow_A^n \langle c_{1A}', mds_A', mem_{1A}'\rangle_A \wedge$
$\langle c_{1A}', mds_A', mem_{1A}'\rangle_A\ \mathcal{R}\ \langle c_{1C}', mds_C', mem_{1C}'\rangle_C \wedge$
$(\forall\, c_{2A}\ mem_{2A}\ c_{2C}\ mem_{2C}\ c_{2A}'\ mem_{2A}'.$
$\langle c_{1A}, mds_A, mem_{1A}\rangle_A\ \mathcal{B}\ \langle c_{2A}, mds_A, mem_{2A}\rangle_A \wedge$
$\langle c_{2A}, mds_A, mem_{2A}\rangle_A\ \mathcal{R}\ \langle c_{2C}, mds_C, mem_{2C}\rangle_C \wedge$
$\langle c_{1C}, mds_C, mem_{1C}\rangle_C\ \mathcal{I}\ \langle c_{2C}, mds_C, mem_{2C}\rangle_C \wedge$
$\langle c_{2A}, mds_A, mem_{2A}\rangle_A \rightsquigarrow_A^n \langle c_{2A}', mds_A', mem_{2A}'\rangle_A \longrightarrow$
$(\exists\, c_{2C}'\ mem_{2C}'.$
$\langle c_{2C}, mds_C, mem_{2C}\rangle_C \rightsquigarrow_C \langle c_{2C}', mds_C', mem_{2C}'\rangle_C \wedge$
$\langle c_{2A}', mds_A', mem_{2A}'\rangle_A\ \mathcal{R}\ \langle c_{2C}', mds_C', mem_{2C}'\rangle_C \wedge$
$\langle c_{1C}', mds_C', mem_{1C}'\rangle_C\ \mathcal{I}\ \langle c_{2C}', mds_C', mem_{2C}'\rangle_C))))$

(a) Formal definition of coupling-inv-pres $\mathcal{B}\ \mathcal{R}\ \mathcal{I}$.



(b) Graphical depiction of coupling invariant preservation, coupling-inv-pres $\mathcal{B}\ \mathcal{R}\ \mathcal{I}$.

Fig. 5: Coupling invariant preservation.

the coupling invariant $\mathcal{I}$, denoted secure-refinement $\mathcal{B}\ \mathcal{R}\ \mathcal{I}$.

secure-refinement $\mathcal{B}\ \mathcal{R}\ \mathcal{I} \equiv$
preserves-modes-mem $\mathcal{R}\ \wedge$ closed-others $\mathcal{R}\ \wedge$
new-vars-private $\mathcal{R}\ \wedge$ cg-consistent $\mathcal{I}\ \wedge$ sym $\mathcal{I}\ \wedge$
coupling-inv-pres $\mathcal{B}\ \mathcal{R}\ \mathcal{I}$

Notice the not-unexpected side-condition (cg-consistent) that $\mathcal{I}$ be preserved by the possible memory-updates performed by other components, and symmetric. Then we have that:

Theorem 5.1:

$$\frac{\text{strong-low-bisim-mm}\ \mathcal{B} \qquad \text{secure-refinement}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I}}{\text{strong-low-bisim-mm}\ (\mathcal{B}_C\text{of}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I})}$$

Proof Sketch We need to prove that $\mathcal{B}_C$of $\mathcal{B}\ \mathcal{R}\ \mathcal{I}$ is symmetric, satisfies cg-consistent, implies Low-equivalence modulo modes, and is preserved by the steps of the concrete component. It is easily shown symmetric, since $\mathcal{B}$ and $\mathcal{I}$ are. It being cg-consistent follows because (1) $\mathcal{B}$ is, (2) $\mathcal{R}$ is closed-others, and (3) $\mathcal{I}$ is cg-consistent. Finally, it is preserved locally by the component largely because it satisfies coupling-inv-pres, which guarantees that for any concrete step, there exists a matching one starting from a related state under $\mathcal{B}_C$of $\mathcal{B}\ \mathcal{R}\ \mathcal{I}$. The steps end in $\mathcal{I}$-related configurations that are $\mathcal{R}$-related to $\mathcal{B}$-related abstract configurations. It remains to show that the two final concrete configurations, labelled $1C'$ and $2C'$ in Fig. 5b, are Low-equivalent modulo modes. Because $\mathcal{R}$ preserves abstract memory contents, this equivalence is guaranteed to hold for all concrete variables that correspond to abstract ones (i.e. those in range var$_C$of). For the remaining variables, new-vars-private ensures that any changes to them or their classifications cannot violate the equivalence, which held between the initial concrete states. ∎

E. Discussion

Like $\mathcal{B}$, $\mathcal{I}$ must be symmetric and cg-consistent; but this is where the similarity to $\mathcal{B}$ ends. Specifically, $\mathcal{I}$ need not imply that memory contents are secure. Its job is merely to ensure that the second initial concrete configuration (labelled $2C$ in Fig. 5b), is *synchronised* with the first ($1C$). To see why, consider the case where an abstract action is refined to multiple concrete ones, as occurs for instance in the refinement of the addition statement in Fig. 2a to Fig. 2b. Then there will exist multiple concrete configurations each related to the same abstract configuration under $\mathcal{R}$, where some are direct predecessors of others under the concrete transition relation $\rightsquigarrow_C$. The coupling invariant's job is to exclude configurations for $2C$ that do not correspond to the same point in time as $1C$. Doing so requires only talking about the location within the component's program to which each configuration corresponds.

For example, let $1C$ be the configuration corresponding to the execution of the second **skip** statement of the **then**-branch of Fig. 2b, and $1A$ be the $\mathcal{R}$-related configuration corresponding to the execution of the statement x := y in Fig. 2a. For illustration, let $2A$ be the $\mathcal{B}$-related configuration corresponding to the addition statement x := y + z in Fig. 2a. Then there are multiple choices for $2C$, that are all $\mathcal{R}$-related to this $2A$: for instance both of the configurations relating to the execution of the assignments to reg1$_C$ and reg2$_C$. However, of these, only the reg2$_C$ assignment statement is synchronised with $1C$, the second **skip** statement of the **then**-branch. Recall from Sec. II-B that the coupling invariant for this example relates the second **skip** statement of the **then**-branch to the reg2$_C$ assignment statement of the **else**-branch. Indeed, the reg2$_C$ assignment is the *only* one that is both related under this coupling invariant to $1C$ (the second **skip** statement), and related under $\mathcal{R}$ to $2A$ (the addition statement). In this way, the coupling invariant makes sure the right choice for $2C$ is made. Its preservation ensures that the concrete program's Low-observable timing behaviour is independent of non-Low data, by ensuring in this case that the execution of the two branches of the **if**-statement remain synchronised.

Note that in the example above there are other possibilities for $2A$, namely those configurations corresponding to the execution of x := y (which, recall, is the statement to which $1A$ corresponds). In this situation, our refinement theory demands proving the triviality that a program's execution is synchronised with itself, and so places the trivial requirement on coupling invariants to relate configurations whose commands are identical.

F. Compositional, Whole-System Refinement

We can of course lift Theorem 5.1 to entire systems, via Theorem 3.1, by proving that the refinement relations (when applied to their respective components) preserve sound-mode-use, the side condition on Theorem 3.1.[2] Recall that this condition requires that: (1) all components guarantee the assumptions of all others, and (2) all components adhere to their own guarantees. (1) is a global property of the entire system, while (2) is a property of each component.

In our Isabelle/HOL formalisation, we consider the case in which the set of all new concrete variables (i.e. those not in range var_C of) is partitioned between the various components, setting the assumptions and guarantees of each component statically to ensure that its new concrete variables remain private to it. In this situation, we prove that (1) is preserved under refinement, by proving that every execution of the concrete system has a corresponding execution in the abstract system. This result is similar in spirit to those for other compositional theories of refinement [LFF14]. From this we derive that the set of reachable memories and mode states for the concrete system corresponds to the same set for the abstract system, from which we derive that (1) is preserved.

Proving that (2) holds for an individual refined component can either be done using an appropriate type system (we defined one based on [MSS11] and proved it sound in our formalisation), or by adding an extra condition on the refinement relations $\mathcal{R}$. Our Isabelle/HOL formalisation defines one such condition and proves it sufficient.

Our formalisation contains the full statement of the preservation of sys-secure under refinement.

VI. Simple Refinement

We now exercise the theory of Sec. V by considering the class of *simple* components that do not branch on non-Low data. By this, we mean formally that their abstract bisimulation relation $\mathcal{B}$ only ever relate configurations $\langle c_{1A}, mds_A, mem_{1A}\rangle_A$ and $\langle c_{2A}, mds_A, mem_{2A}\rangle_A$ for which $c_{1A} = c_{2A}$. We denote this property bisim-simple $\mathcal{B}$ and use the term "simple" to refer interchangeably to the bisimulation $\mathcal{B}$ and the abstract component it proves secure.

bisim-simple $\mathcal{R}_A \equiv$
$\forall\, c_{1A}\ mds\ mem_{1A}\ c_{2A}\ mem_{2A}.$
$(\langle c_{1A}, mds, mem_{1A}\rangle_A, \langle c_{2A}, mds, mem_{2A}\rangle_A) \in \mathcal{R}_A \longrightarrow$
$c_{1A} = c_{2A}$

[2] An alternative would be to prove it directly for the refined system [MMOPW15], or have it enforced using dynamic monitoring [ACM15].

The Low-observable timing behaviour of simple components depends only on Low data. Recall that the purpose of the coupling invariant $\mathcal{I}$ is to ensure this kind of timing-independence. Thus one might expect that, for such components, we can securely refine them under a much simpler notion of refinement without need of a coupling invariant.

We prove this true. We define a much simpler notion of secure refinement, secure-refinement-simple. Then we define a very simple coupling invariant, $\mathcal{I}_{\mathsf{simple}}$, and prove that secure-refinement-simple implies secure-refinement with $\mathcal{I}_{\mathsf{simple}}$.

secure-refinement-simple has no coupling invariant and, of the conditions depicted in Fig. 5b, requires proving just those that correspond to the front face of the cube. However, it also places some side-conditions on the refinement relation $\mathcal{R}$ that make explicit the requirement that it is not allowed to introduce branching on non-Low data, nor have the Low-observable timing properties of the refinement depend on non-Low data. These side conditions are in many ways far more intuitive than those of coupling-inv-pres from Sec. V.

To satisfy these side conditions one must supply a function *abs-steps* that, given a pair of $\mathcal{R}$-related abstract and concrete configurations, returns a natural number n that is the number of abstract execution steps that need to be performed to match a single concrete execution step from the given respective configurations. We use *abs-steps* to assert that the timing behaviour of the refined component depends only on Low data.

We collect the side conditions in a predicate that we denote simple-refinement-safe, parameterised by *abs-steps*:

simple-refinement-safe $\mathcal{B}\ \mathcal{R}$ *abs-steps* $\equiv$
$\forall\, c_A\ mds_A\ mem_{1A}\ mem_{2A}\ c_C\ mds_C\ mem_{1C}\ mem_{2C}.$
$\langle c_A, mds_A, mem_{1A}\rangle_A\ \mathcal{B}\ \langle c_A, mds_A, mem_{2A}\rangle_A \wedge$
$\langle c_A, mds_A, mem_{1A}\rangle_A\ \mathcal{R}\ \langle c_C, mds_C, mem_{1C}\rangle_C \wedge$
$\langle c_A, mds_A, mem_{2A}\rangle_A\ \mathcal{R}\ \langle c_C, mds_C, mem_{2C}\rangle_C \longrightarrow$
stops $\langle c_C, mds_C, mem_{1C}\rangle_C =$ stops $\langle c_C, mds_C, mem_{2C}\rangle_C \wedge$
abs-steps $\langle c_A, mds_A, mem_{1A}\rangle_A\ \langle c_C, mds_C, mem_{1C}\rangle_C =$
abs-steps $\langle c_A, mds_A, mem_{2A}\rangle_A\ \langle c_C, mds_C, mem_{2C}\rangle_C \wedge$
$(\forall\, mds_{1C}'\ mds_{2C}'\ mem_{1C}'\ mem_{2C}'\ c_{1C}'\ c_{2C}'.$
$\langle c_C, mds_C, mem_{1C}\rangle_C \rightsquigarrow_C \langle c_{1C}', mds_{1C}', mem_{1C}'\rangle_C \wedge$
$\langle c_C, mds_C, mem_{2C}\rangle_C \rightsquigarrow_C \langle c_{2C}', mds_{2C}', mem_{2C}'\rangle_C \longrightarrow$
$c_{1C}' = c_{2C}' \wedge mds_{1C}' = mds_{2C}')$

It considers two concrete configurations with identical commands and mode states (but possibly differing memories), that are $\mathcal{R}$-related to a pair of $\mathcal{B}$-related abstract configurations. Thus the two concrete configurations disagree only on non-Low memory locations. It asserts that the two configurations must agree on (1) whether the concrete component terminates, and (2) how long it takes to execute, and, thus, asserts that these properties cannot depend on non-Low data. The final outermost conjunct requires that $\mathcal{R}$ never introduce new branching on non-Low data.

Our simpler notion of refinement, which corresponds to just the front face of the cube in Fig. 5b, we denote simple-refinement-pres. Its formal definition appears in Fig. 6.

We can now define secure-refinement-simple.

$$
\begin{aligned}
&\mathsf{simple\text{-}refinement\text{-}pres}\ \mathcal{R}\ \mathit{abs\text{-}steps} \equiv \\
&\forall\, c_{1A}\ \mathit{mds}_A\ \mathit{mem}_{1A}\ c_{1C}\ \mathit{mds}_C\ \mathit{mem}_{1C}\ n. \\
&\quad \langle c_{1A}, \mathit{mds}_A, \mathit{mem}_{1A}\rangle_A\ \mathcal{R}\ \langle c_{1C}, \mathit{mds}_C, \mathit{mem}_{1C}\rangle_C \wedge \\
&\quad n = \\
&\quad \mathit{abs\text{-}steps}\ \langle c_{1A}, \mathit{mds}_A, \mathit{mem}_{1A}\rangle_A\ \langle c_{1C}, \mathit{mds}_C, \mathit{mem}_{1C}\rangle_C \longrightarrow \\
&\quad (\forall\, c_{1C}'\ \mathit{mds}_C'\ \mathit{mem}_{1C}'. \\
&\qquad \langle c_{1C}, \mathit{mds}_C, \mathit{mem}_{1C}\rangle_C \rightsquigarrow_C \langle c_{1C}', \mathit{mds}_C', \mathit{mem}_{1C}'\rangle_C \longrightarrow \\
&\qquad (\exists\, c_{1A}'\ \mathit{mds}_A'\ \mathit{mem}_{1A}'. \\
&\qquad\quad \langle c_{1A}, \mathit{mds}_A, \mathit{mem}_{1A}\rangle_A \rightsquigarrow_A^n \langle c_{1A}', \mathit{mds}_A', \mathit{mem}_{1A}'\rangle_A \wedge \\
&\qquad\quad \langle c_{1A}', \mathit{mds}_A', \mathit{mem}_{1A}'\rangle_A\ \mathcal{R}\ \langle c_{1C}', \mathit{mds}_C', \mathit{mem}_{1C}'\rangle_C))
\end{aligned}
$$

Fig. 6: Simple refinement, for simple components.

$$
\begin{aligned}
&\mathsf{secure\text{-}refinement\text{-}simple}\ \mathcal{B}\ \mathcal{R}\ \mathit{abs\text{-}steps} \equiv \\
&\mathsf{preserves\text{-}modes\text{-}mem}\ \mathcal{R} \wedge \mathsf{closed\text{-}others}\ \mathcal{R} \wedge \\
&\mathsf{new\text{-}vars\text{-}private}\ \mathcal{R} \wedge \mathsf{simple\text{-}refinement\text{-}safe}\ \mathcal{B}\ \mathcal{R}\ \mathit{abs\text{-}steps} \wedge \\
&\mathsf{simple\text{-}refinement\text{-}pres}\ \mathcal{R}\ \mathit{abs\text{-}steps}
\end{aligned}
$$

Let $\mathcal{I}_{\mathsf{simple}}$ be the coupling invariant that requires (only) that the commands of the two concrete configurations agree.

$$
\begin{aligned}
&\mathcal{I}_{\mathsf{simple}} \equiv \\
&\{(\langle c_{1C}, \mathit{mds}_{1C}, \mathit{mem}_{1C}\rangle_C, \langle c_{2C}, \mathit{mds}_{2C}, \mathit{mem}_{2C}\rangle_C) \mid c_{1C} = c_{2C}\}
\end{aligned}
$$

Note that $\mathcal{I}_{\mathsf{simple}}$ is trivially symmetric and satisfies cg-consistent, since it doesn't talk at all about memory. Then:

Theorem 6.1:

$$
\frac{\mathsf{bisim\text{-}simple}\ \mathcal{B} \qquad \mathsf{secure\text{-}refinement\text{-}simple}\ \mathcal{B}\ \mathcal{R}\ \mathit{abs\text{-}steps}}{\mathsf{secure\text{-}refinement}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I}_{\mathsf{simple}}}
$$

VII. Related Work

We compare our two main contributions separately to the most closely related work.

A. Dependent Type Systems

Dependent security type systems have become increasingly well studied of late. To our knowledge, Zheng and Myers [ZM04], [ZM07] proposed the first dependent security type system for dealing with dynamic changes to runtime security labels in the context of Jif. A number of functional languages have been developed with dependent type systems, used to encode value-dependent information flow properties, e.g. Fine [SCC10] and its successor F* [SCF+11].

In the context of imperative languages, Deputy [CHA+07] and Xanadu [Xi00] both incorporate dependent type systems. Here the focus is on execution safety rather than enforcing noninterference.

The only dependent security type system we are aware of for a concurrent software language is the very recent work of Li et al. [LNNF16], who present a dependent type system for a form of value-dependent noninterference for a language with concurrent components that communicate by message-passing, similar to a process calculus. Here each component has its own private memory, but components do not share memory.

Zhang et al. also recently developed the hardware design language SecVerilog [ZWSM15], which incorporates dependent security types for enforcing timing-sensitive information flow security and allows describing hardware modules that operate in parallel to each other.

In contrast, our type system is the first we are aware of for compositionally reasoning about concurrent programs that share memory.

B. Refinement

Refinement is relatively well studied in the context of noninterference. Jacob first pointed out that noninterference is not always preserved by refinement [Jac88], a result since referred to as the *refinement paradox*.

Approaches to preserving noninterference under refinement have included strengthening the noninterference property sufficiently so that it is guaranteed to always be preserved by refinement [Low07], or restricting the definition of refinement to preserve noninterference [Man01].

In the context of preserving noninterference for concurrent systems under compositional refinement, where each component is refined individually, [ML09] consider this question in the context of the process algebra CSP [Hoa85], in the absence of shared memory. Our theory is the first we are aware of that considers preserving timing-sensitive noninterference for concurrent shared memory programs under compositional refinement.

Our approach is heavily influenced by the RGSim framework [LFF14], a compositional refinement framework aimed at preserving ordinary safety properties and, in later work liveness ones too [LFS14], but not noninterference. Our framework effectively shows how to strengthen compositional refinement in order to preserve timing-sensitive noninterference.

VIII. Conclusion

We have presented the first dependent type system for compositionally verifying timing-sensitive, value-dependent noninterference for concurrent programs with shared-memory. Our type system deeply embeds types as sets of predicates, so that type comparisons are phrased in terms of predicate entailment, to ease automation. It tracks a set of predicates about what is known to be true at the current point in the program, and its rules support rewriting of the typing context to deal with the possible modifications to variables by other threads.

We also presented the first theory of compositional refinement for preserving timing-sensitive noninterference for concurrent, shared-memory programs when refining each of the program's threads one-at-a-time. It makes use of coupling invariants to prove that the refinement relation correctly preserves the constant-time nature of the original program. We showed that for simple programs that do not branch on confidential data, one can dispense with coupling invariants by instead proving directly that the refinement relation neither introduces publicly observable timing dependencies on confidential data, nor new branching on confidential data.

Our work here rests on an underlying foundation [Mur15], [MSS11] for reasoning about noninterference for concurrent programs rooted in assume-guarantee style reasoning [Jon81].

The connection between this style of reasoning and concurrent separation logic (CSL) is well-known [FFS07], [VP07]. An interesting direction for future work would be to investigate the use of CSL for compositionally reasoning about value-dependent noninterference in the presence of concurrency.

We hope that the results presented in this paper will underpin the future development of verified concurrent programming languages for enforcing expressive information flow policies, particularly in an age when data-intensive applications, with data-dependent security policies, are becoming increasingly popular.

Acknowledgements

We thank the anonymous reviewers for their insightful feedback.

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

[ABB06] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *Thirty-third Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 91–102, 2006.

[ACM15] Aslan Askarov, Stephen Chong, and Heiko Mantel. Hybrid monitors for concurrent noninterference. In *28th IEEE Computer Security Foundations Symposium (CSF)*, pages 137–151, 2015.

[BC02] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1-2):109–130, 2002.

[CHA+07] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C Necula. Dependent types for low-level programming. In *16th European Symposium on Programming Languages and Systems (ESOP)*, pages 520–535. 2007.

[FFS07] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, pages 173–188, 2007.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. 1985.

[Jac88] Jeremy Jacob. Security specifications. In *IEEE Symposium on Security and Privacy*, pages 14–23, 1988.

[Jon81] Cliff B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. D.Phil. thesis, University of Oxford, June 1981.

[LC15] Luísa Lourenço and Luís Caires. Dependent information flow types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 317–328, Mumbai, India, January 2015.

[LFF14] Hongjin Liang, Xinyu Feng, and Ming Fu. Rely-guarantee-based simulation for compositional verification of concurrent program transformations. *ACM Transactions on Programming Languages and Systems*, 36(1):3:1–3:55, March 2014.

[LFS14] Hongjin Liang, Xinyu Feng, and Zhong Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *CSL-LICS*, July 2014.

[LNNF16] Ximeng Li, Flemming Nielson, Hanne Riis Nielson, and Xinyu Feng. Disjunctive information flow for communicating processes. In *TGC 2016*, volume 9533 of *Lecture Notes in Computer Science*, pages 95–111, 2016.

[Low07] Gavin Lowe. On information flow and refinement-closure. In *Proceedings of the 7th Workshop on Issues in the Theory of Security*, March 2007.

[Man01] Heiko Mantel. Preserving information flow properties under refinement. In *IEEE Symposium on Security and Privacy*, pages 78–91, 2001.

[ML09] Toby Murray and Gavin Lowe. On refinement-closed security properties and nondeterministic compositions. In *Proceedings of the 8th International Workshop on Automated Verification of Critical Systems*, volume 250 of *Electronic Notes in Theoretical Computer Science*, pages 49–68, Glasgow, UK, 2009.

[MMB+12] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Noninterference for operating system kernels. In *International Conference on Certified Programs and Proofs*, pages 126–142, Kyoto, Japan, December 2012.

[MMOPW15] Heiko Mantel, Markus Müller-Olm, Matthias Perner, and Alexander Wenner. Using dynamic pushdown networks to automate a modular information-flow analysis. In *25th International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR)*, 2015.

[MMW16] Daniel Matichuk, Toby Murray, and Makarius Wenzel. Eisbach: A proof method language for Isabelle. *Journal of Automated Reasoning*, 56(3):261–282, 2016.

[MSS11] Heiko Mantel, David Sands, and Henning Sudbrock. Assumptions and guarantees for compositional noninterference. In *IEEE Computer Security Foundations Symposium*, pages 218–232, Cernay-la-Ville, France, June 2011.

[Mur15] Toby Murray. On high-assurance information-flow-secure programming languages. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 43–48, Prague, Czech Republic, July 2015.

[MWM14] Daniel Matichuk, Makarius Wenzel, and Toby Murray. An Isabelle proof method language. In *International Conference on Interactive Theorem Proving*, pages 390–405, Vienna, Austria, July 2014.

[NBG11] A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *IEEE Symposium on Security and Privacy*, pages 165–179, May 2011.

[NPW02] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. 2002.

[PN02] Leonor Prensa Nieto. *Verification of parallel programs with the Owicki-Gries and rely-guarantee methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2002.

[SBN13] Gordon Stewart, Anindya Banerjee, and Aleksandar Nanevski. Dependent types for enforcement of information flow and erasure policies in heterogeneous data structures. In *Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 145–156, 2013.

[SCC10] Nikhil Swamy, Juan Chen, and Ravi Chugh. Enforcing stateful authorization and information flow policies in Fine. In *European Symposium on Programming (ESOP)*, March 2010.

[SCF+11] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *16th ACM SIGPLAN international conference on Functional Programming (ICFP)*, pages 266–278, 2011.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[SS00] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *13th IEEE Computer Security Foundations Workshop (CSFW)*, pages 200–215, 2000.

[VP07] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.

[VS99] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, 1999.

[Xi00] Hongwei Xi. Imperative programming with dependent types. In *15th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 375–387, 2000.

[ZM04] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. In *Formal Aspects in Security and Trust (FAST)*, pages 27–40, 2004.

[ZM07] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2–3), March 2007.

[ZWSM15] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *ASPLOS*, 2015.