Short Paper: On High-Assurance Information-Flow-Secure Programming Languages

Toby Murray
NICTA and the University of New South Wales
Sydney, Australia
toby.murray@nicta.com.au

ABSTRACT

We argue that high-assurance systems require high-assurance information-flow-secure programming languages. As a step towards such languages, we present the, to our knowledge, first concurrent theory of information flow security that supports (1) compositional reasoning under dynamic assumptions, and (2) value-dependent classification, to handle the dynamism inherent in modern high-assurance systems. We sketch out our vision and a roadmap for building self-certifying information-flow-secure programming languages.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General—*Protection mechanisms*; D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*

General Terms

Languages, Security, Theory, Verification

1. INTRODUCTION

Information flow security has remained an active topic of research since the seminal work of Denning [1976] and Goguen and Meseguer [1982]. Much of this early work sought to develop theories for proving the absence of unwanted information leakage in *high-assurance systems*, like those that process classified data. Decades later, these systems are more prevalent and no less security-critical. Despite facing greater security threats, modern security-critical systems are rarely formally proved to be information-flow secure, not least because doing so remains fairly expensive [Murray et al. 2013].

Meanwhile, of all the information flow security research, information-flow-secure programming languages like Jif [Myers 1999], JSFlow [Hedin et al. 2014] and Paragon [Broberg et al. 2013], have arguably emerged as the best vehicles for putting security theory [Sabelfeld and Myers 2003] into the hands of programmers for constructing secure systems [Chong et al. 2007; Clarkson et al. 2008]. They allow programmers to express information flow guarantees that are then enforced by the compiler through some combination of static and dynamic checks. Importantly, the programmer need not understand how the checks work, nor why they are sound, so long as the compiler is trusted.

Thus one way to build high-assurance systems, with formally verified security at reasonable cost, might be to use information-flow-secure programming languages. However, current languages are ill-suited to high-assurance systems because each has an overly large trusted computing base (TCB). For instance, Jif and Paragon rely on Java, so their TCB includes not only their compiler but also the Java TCB — which in 2002 comprised anywhere upwards of 50,000 to 230,000 lines of unverified code [Appel and Wang 2002].

We argue that high-assurance systems demand high-assurance information-flow-secure programming languages. The compiler for such a language should not have to be trusted. Instead, the output it produces should be automatically formally certified as being secure. Recognising that security is the overriding concern for these systems, such a language can also eschew general-purpose language features to reduce the size of its TCB, and simplify the job of certifying its compiler-produced output [Keller et al. 2013].

Such languages must also be able to handle the concurrency and dynamism of modern high-assurance systems. Consider a *dual-personality* smartphone whose *classified* personality allows the user to send and receive classified information that the phone guarantees will never be observable outside this personality. The classification of the input that the user types into the phone thus changes dynamically, depending on which personality is active. The *input driver* component, which receives user input and runs concurrently to the two phone personalities, is required to copy incoming input to (only) the currently active personality. We need languages that allow the programmer to naturally express this kind of dynamic policy and enforce a suitably dynamic, concurrent notion of information-flow security. Additionally, if the compiler output is to be automatically formally certified, then the language requires a *compositional* theory of information flow security to allow each concurrent component to be certified independently; otherwise, it won't scale.

As a first step towards this vision, we present the first compositional theory of concurrent, value-dependent information flow security, able to accommodate this kind of dynamism. This theory lays the foundation for the development of self-certifying high-assurance information-flow-secure programming languages, for which we provide a roadmap.

2. A MOTIVATING EXAMPLE

To motivate our new compositional theory of information flow security, consider the pseudocode in Figure 1. It is a fragment of

PLAS'15 July 06 2015, Prague, Czech Republic

ACM 978-1-4503-3661-1/15/07...$15.00
DOI: http://dx.doi.org/10.1145/2786558.2786561 .

a simplified input driver component, inspired by the example of Section 1. It reads input, from a variable `input`, and copies it via the `temp` variable to one of two *input buffer* variables, `low` and `high`, depending on which personality is active, stored in the `cur_pers` variable.

The `input` variable is updated by some other concurrently running component whenever new input is available; likewise, the `cur_pers` variable is updated when the user switches between the two system personalities.

Here, the classification of the data held by the `input` variable varies dynamically. At any point in time, its classification is determined by the `cur_pers` variable: `input` is classified Low iff `cur_pers` is zero, and is High otherwise. Thus `input`'s classification is *value-dependent* [Zheng and Myers 2007; Lourenço and Caires 2015].

The comments encode *assumptions* that this code makes to be correct. It assumes that no other component can modify the `input` variable, nor modify or read the `temp` variable that temporarily holds the input (and does so *before* the classification of the `input` variable is learned). The assumption that other components won't modify `input` while this code is running implies also that they won't change `input`'s classification by modifying `cur_pers`. The assumption that `temp` is not read by other components means it is safe to be classified Low always [Mantel et al. 2011]. In practice, these assumptions would be enforced using appropriate concurrency control primitives, like locks.

```
1 // assume: NoWrite input
2 // assume: NoReadOrWrite temp
3 temp = input;
4 if (cur_pers == 0)
5    low = temp;
6 else
7    high = temp;
8 temp = 0; // clear temp
```

Figure 1: A snippet of a dynamic input driver component.

3. COMPOSITIONAL VALUE-DEPENDENT INFORMATION FLOW SECURITY

We require a theory of information flow security that supports concurrency via *compositional reasoning*, to allow each component to be automatically proved secure independently, under (dynamic) *assumptions* they make about the behaviour of other components, while supporting *value-dependent* classification. To construct a prototype theory, we extended the theory of Mantel et al. [2011], which provides for compositional reasoning with assumptions, to incorporate a notion of value-dependent classification.

To our knowledge, this is the first such theory that incorporates all of these elements. Previous value-dependent theories (e.g. [Zheng and Myers 2007; Lourenço and Caires 2015]) are not concurrent, or (e.g. [Murray et al. 2012]) don't support compositional reasoning.

To extend the theory of Mantel et al. [2011] to include value-dependent classification, we draw on ideas from Murray et al. [2012] who extended the *noninfluence* theory of von Oheimb [2004] to allow the classification of actions to be state-dependent. This was achieved by (1) carefully delineating the part of the state on which dynamic classifications depended, and (2) ensuring that this state was always classified at the lowest level (i.e. was public knowledge). Thus in a secure system the state that determines the dynamic classifications can never be influenced by confidential information, and thus dynamic changes to classifications cannot form a covert channel. We reuse these same ideas.

Our theory is formalised and its central compositionality theorem proved in the Isabelle/HOL proof assistant [Nipkow et al. 2002], by modifying the existing formalisation by Grewe et al. [2014] of the theory of Mantel et al. [2011].

3.1 Preliminaries

Memory is modelled as a mapping from a finite set of *variables* (memory addresses) to *values*. As is usual, we restrict our attention to a two-point lattice of security classifications High and Low, where Low $<$ High. Let $\mathcal{L}_{mem}\ v$ give the classification of variable v when the memory is mem. $\mathcal{L}$ is parameterised by mem to accommodate variables whose classification depends on the values of other variables, like `input` in Figure 1. Let $\mathcal{C}$vars v denote the (fixed) set of variables that variable v's classification depends on, such that:

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

Let $\mathcal{C} \equiv \bigcup_x \mathcal{C}\text{vars}\ x$ 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: $\forall x{\in}\mathcal{C}.\ \mathcal{L}_{mem}\ x = \text{Low}$ and $\forall x{\in}\mathcal{C}.\ \mathcal{C}\text{vars}\ x = \emptyset$.

In the example of Figure 1, $\mathcal{C}$vars `input` $= \{$`cur_pers`$\}$, and $\mathcal{C}$vars $v = \emptyset$ for all other variables v. The requirement then that `cur_pers` is always classified Low simply encodes the necessary assumption that the user's choices about which personality is currently active do not themselves leak High information (i.e. that the user isn't an unwitting covert channel). Regardless of whether this assumption is valid in practice, the system is insecure without it.

Following [Mantel et al. 2011], we assume a deterministic programming language in which each concurrently executing component is 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, via *assume-guarantee*-style reasoning [Jones 1981]. Let AsmNoWrite, AsmNoReadOrWrite, GuarNoWrite, and GuarNoReadOrWrite denote four (not mutually exclusive) *modes* that a component may dynamically associate with each variable. When a component associates the AsmNoWrite mode with a variable v, the component assumes that no other component will modify v or its classification (by modifying its $\mathcal{C}$vars). The AsmNoReadOrWrite mode assumes additionally that the variable and its $\mathcal{C}$vars (i.e. its classification) will not be read. When a component associates GuarNoWrite with variable v, it is guaranteeing not to modify v nor its classification; GuarNoReadOrWrite additionally guarantees that v and its $\mathcal{C}$vars will not be read.

The mode state is a mapping from each mode to the set of variables that currently have that mode. Thus variable v has e.g. mode AsmNoWrite in mode state mds when $v \in mds$ AsmNoWrite. The mode state acts as *ghost state*, enriching the language semantics with sufficient information to allow compositional reasoning; however this information does not affect memory contents. In the example of Figure 1, the two comments are annotations that update the mode state by associating `input` and `temp` with the modes AsmNoWrite and AsmNoReadOrWrite respectively.

AsmNoReadOrWrite and GuarNoReadOrWrite are a slight departure from Mantel et al. [2011] who have instead an assumption and guarantee that forbids reading a variable while allowing it to be modified. In our experience, this situation doesn't tend to arise in practice; however, accommodating it is a significant source of complexity in Grewe et al. [2014]. Our change simplifies the theory without a practical loss of applicability, allowing us to concentrate on the new aspects of value-dependent classification.

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. $(cms, mem) \rightsquigarrow_i (cms', mem')$ denotes that the system transitions from global configuration (cms, mem) to configuration (cms', mem') by the ith component making an execution step. It is defined as follows.

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

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

We abstract away from any particular scheduling policy or implementation by defining execution against arbitrary *schedules* as follows. A schedule $sched$ is a finite list of natural numbers, prescribing the order in which components are to execute. Execution against $sched$ is denoted $\rightarrow_{sched}$, which is a transition relation on global configurations defined recursively in the natural way.

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

Here $[]$ is the empty list and $n \cdot ns$ is the list whose head is n and whose tail is the list ns.

3.2 Security

We now define the main security properties. They naturally parallel the original definitions of Mantel et al. [2011], wherein 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 via assume-guarantee style reasoning [Jones 1981].

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: $mem_1 =^l mem_2 \longrightarrow (\forall x.\ \mathcal{L}_{mem_1}\ x = \mathcal{L}_{mem_2}\ x)$.

Let mds_0 denote the *initial* mode state: $\mathsf{mds}_0\ m = \emptyset$. For a list $cmds = [cmd_1, \ldots, cmd_n]$ of commands, let init $cmds$ be the list: $[(cmd_1, \mathsf{mds}_0), \ldots, (cmd_n, \mathsf{mds}_0)]$. For a list x let set x denote the set containing just x's elements.

Finally, let sys-secure $cmds$ be the global security property that denotes when the collection of concurrently executing components $cmds$ is secure:

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

Here modes-eq $cms_1'\ cms_2'$ denotes that the two lists cms_1' and cms_2' agree pointwise on their mode states; readable $cms_1'\ x \equiv \forall (cmd', mds') \in \textsf{set}\ cms_1'.\ x \notin mds'$ AsmNoReadOrWrite.

sys-secure $cmds$ 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.

By quantifying over schedules $sched$, our global security property effectively assumes that the operation of the system scheduler is determined by a static schedule that is public knowledge. The same assumption is made for instance in the seL4 information flow security proof [Murray et al. 2013]. Note that this quantification over schedules is not present in the original property of Mantel et al. [2011] and makes our global security property slightly stronger.

Local Security.

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

$$\begin{array}{l}
mem_1 =_{mds}{}^l\ mem_2 \equiv \\
\forall x.\ x \in \mathcal{C} \vee \\
\quad \mathcal{L}_{mem_1}\ x = \mathsf{Low} \wedge x \notin mds\ \textsf{AsmNoReadOrWrite} \longrightarrow \\
\quad mem_1\ x = mem_2\ x
\end{array}$$

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, we require that for each a relation $\mathcal{R}$ can be found that relates two executions of the component and ensures that the Low-equivalence modulo modes is always preserved. Following Mantel et al. [2011], $\mathcal{R}$ is called a *strong low bisimulation modulo modes* and is defined formally as follows.

We require that $\mathcal{R}$ is preserved by the actions of the other components in the system, restricted according to the assumptions encoded in the current mode state mds. In this case we say that $\mathcal{R}$ is *closed under globally consistent changes*, denoted closed-gc $\mathcal{R}$.

$$\begin{array}{l}
\textsf{closed-gc}\ \mathcal{R} \equiv \\
\forall c_1\ mds\ mem_1\ c_2\ mem_2. \\
\quad \langle c_1, mds, mem_1\rangle\ \mathcal{R}\ \langle c_2, mds, mem_2\rangle \longrightarrow \\
\quad (\forall A.\ (\forall x.\ mem_1\ x \neq mem_1\ [\|_1\ A]\ x \vee \\
\qquad\qquad mem_2\ x \neq mem_2\ [\|_2\ A]\ x \longrightarrow \\
\qquad\qquad \textsf{writable}\ mds\ x) \wedge \\
\qquad (\forall x.\ \mathcal{L}_{mem_1}\ x \neq \mathcal{L}_{mem_1\ [\|_1\ A]}\ x \longrightarrow \\
\qquad\qquad \textsf{writable}\ mds\ x) \wedge \\
\qquad mem_1\ [\|_1\ A] =_{mds}{}^l\ mem_2\ [\|_2\ A] \longrightarrow \\
\qquad \langle c_1, mds, mem_1\ [\|_1\ A]\rangle\ \mathcal{R}\ \langle c_2, mds, mem_2\ [\|_2\ A]\rangle)
\end{array}$$

closed-gc $\mathcal{R}$ 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). 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. We restrict A to only modify the values or classifications of variables x that are assumed to be writable: writable $mds\ x \equiv x \notin mds$ AsmNoWrite $\wedge\ x \notin mds$ AsmNoReadOrWrite. We also restrict it to preserving Low-equivalence modulo modes, assuming that all other components behave securely.

We phrase closed-gc in terms of actions A that may modify more than one variable at a time, in contrast to Mantel et al. [2011] who consider only individual variable updates, because we need to take into account how updates to $\mathcal{C}$-variables interact with updates to ordinary variables.

Let strong-low-bisim-mm $\mathcal{R}$ denote that $\mathcal{R}$ is a strong low bisimulation modulo modes [Mantel et al. 2011]:

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

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

Then two commands cmd_1 and cmd_2 are Low-indistinguishable under modes mds, denoted $cmd_1 \sim_{mds} cmd_2$ when:

$$\begin{array}{l}cmd_1 \sim_{mds} cmd_2 \equiv \\ \forall\, mem_1\ mem_2. \\ \quad mem_1 =_{mds}{}^{l}\ mem_2 \longrightarrow \\ \quad (\exists\, \mathcal{R}.\ \text{strong-low-bisim-mm } \mathcal{R} \wedge \\ \qquad \langle cmd_1, mds, mem_1\rangle\ \mathcal{R}\ \langle cmd_2, mds, mem_2\rangle)\end{array}$$

Finally let com-secure cmd be the local security property that denotes when a single component whose program is cmd is secure [Mantel et al. 2011], namely when it is Low-indistinguishable to itself under the initial mode state mds_0:

$$\text{com-secure } cmd \equiv cmd \sim_{\mathsf{mds}_0} cmd$$

Note that, via the compositionality theorem presented shortly, this local security property can be viewed as a proof technique for the global security property sys-secure. Thus we, following Mantel et al. [2011] and many others, effectively use bisimulation as a proof technique for global security.

3.3 Compositionality

The compositionality theorem parallels Mantel et al. [2011]:

$$\textsc{Composition:}\ \frac{\forall\, cmd \in \text{set } cmds.\ \text{com-secure } cmd \qquad \forall\, mem.\ \text{sound-mode-use } (\text{init } cmds, mem)}{\text{sys-secure } cmds}$$

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 (init $cmds, mem$).

sound-mode-use parallels the original [Mantel et al. 2011], so we discuss it only briefly. It 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 AsmNoReadOrWrite (respectively AsmNoWrite) assumption for a variable v, that all other components have GuarNoReadOrWrite (resp. GuarNoWrite) for v. (2) requires that whenever a component whose current command is cmd has the GuarNoReadOrWrite (resp. GuarNoWrite) guarantee for variable v, then condition doesnt-read-or-write $cmd\ v$ (resp. doesnt-write $cmd\ v$) holds.[1]

$$\begin{array}{l}\text{doesnt-read-or-write } cmd\ v \equiv \\ \forall\, mds\ mem\ c'\ mds'\ mem'. \\ \quad \langle cmd, mds, mem\rangle \rightsquigarrow \langle c', mds', mem'\rangle \longrightarrow \\ \quad (\forall\, v' \in \{v\} \cup \mathcal{C}\text{vars } v. \\ \qquad \forall\, x.\ \langle cmd, mds, mem(v' := x)\rangle \rightsquigarrow \langle c', mds', mem'(v' := x)\rangle)\end{array}$$

$$\begin{array}{l}\text{doesnt-write } cmd\ v \equiv \\ \forall\, mds\ mem\ cmd'\ mds'\ mem'. \\ \quad \langle cmd, mds, mem\rangle \rightsquigarrow \langle cmd', mds', mem'\rangle \longrightarrow \\ \quad mem\ v = mem'\ v \wedge \mathcal{L}_{mem}\ v = \mathcal{L}_{mem'}\ v\end{array}$$

(2) asserts this requirement for all *locally reachable* configurations of a component, which are all local configurations reachable through $(\rightsquigarrow \cup \rightsquigarrow_{\text{other}})^*$ where $\rightsquigarrow_{\text{other}}$ captures the steps of other components and is defined inductively as:

$$\frac{\forall\, x.\ \neg\ \text{writable } mds\ x \longrightarrow mem\ x = mem'\ x \wedge \mathcal{L}_{mem}\ x = \mathcal{L}_{mem'}\ x}{\langle c, mds, mem\rangle \rightsquigarrow_{\text{other}} \langle c, mds, mem'\rangle}$$

COMPOSITION *Proof.*

The proof of the compositionality theorem parallels the original, and involves establishing a relational invariant that posits the existence of a set of hypothetical memories (one for each component) that agree with the actual memories except for non-$\mathcal{C}$ variables that are Low and assumed will not be read; it is these hypothetical memories that the invariant asserts are Low-equivalent modulo modes. To complete the proof, some care is needed to construct the hypothetical memories appropriately in light of potential changes to $\mathcal{C}$ variables. The requirement that the AsmNoWrite assumption also prevent modifying a variable's classification (and similarly for AsmNoReadOrWrite and reading) assists the proof, as well as making systems easier to program as illustrated in Figure 1.

4. THE ROAD TO HIGH-ASSURANCE INFORMATION-FLOW-SECURE LANGUAGES

We argue that the time of high-assurance information-flow-secure programming languages has arrived. Specifically, recent advances in verification technology and theory mean that we are now at the point where self-certifying high-assurance languages are within reach. We sketch out what such a language might look like, and how to build it.

High-Assurance Secure Languages.

The compiler for a high-assurance language for programming secure systems should do two things. Firstly, it should automatically prove that the system's source code is secure. Secondly, it should certify that the compiler-produced output correctly implements the source code semantics. When the security property being proved

[1]Note that while the definition of doesnt-read-or-write $cmd\ v$ considers individual variables in $\{v\} \cup \mathcal{C}$vars v, it is equivalent to one that considers arbitrary subsets of $\{v\} \cup \mathcal{C}$vars v.

is preserved by refinement, this ensures that the compiler-produced output is also secure.

Traditionally [Sabelfeld and Myers 2003], information flow security type systems have been used to automatically establish information flow security at the source code semantics. The vast majority of existing information-flow-secure languages have a *type soundness theorem*, which proves that well-typed programs are information flow secure, formalised for an ideal core of the language. However, the compiler for a high-assurance information flow secure language should produce a *proof* of well-typedness for the input program, checked by a trustworthy proof assistant like Isabelle or Coq.

With the advent of CompCert [Leroy 2009], the certified C compiler, compilers that can certify their own output have become a recent reality. A high-assurance language should do no less, in order to remove the compiler from the TCB.

We argue that both of these tasks are feasible, and that they can be combined to produce a self-certifying information-flow-secure programming language. Further, if these proofs are then composed with those of a verified kernel on which the compiled code is deployed, the resulting proof chain offers the hope of truly *end-to-end* proofs of security, all the way from the source code, down through its compiled implementation and the kernel on which it runs.

Source Code Proofs of Security.

The theory presented in Section 3 allows a system to be proved secure, in terms of its source code semantics, one-component-at-a-time, under assumptions made by each component about the others.

We argue that the necessary ingredients now exist for building an appropriate security type system for this theory. Such a type system needs to make use of each component's assumptions when proving well-typedness of that component. However, it also needs to be able to handle value-dependent classification. When presenting their original theory that we extended in Section 3, Mantel et al. [2011] presented a flow-sensitive type system for proving the security of individual components while making use of assumptions. Lourenço and Caires [2015] recently presented a general theory of dependent security types. It remains to be seen how to reconcile the two, and produce a certifying type checker.

Proofs of Implementation Correctness.

While certified compilers like CompCert are now a reality, they remain very expensive to develop. Keller et al. [2013] argue that building a language with a self-certifying compiler is simplified by having the language eschew general-purpose facilities, which are instead provided by application-specific abstract data types implemented and verified outside of the language.

Existing work on certified compilers covers non-concurrent code. Extending it to concurrent systems requires a compositional *refinement* theory for proving that the compiled code correctly implements the source code semantics one-component-at-a-time. Ideally, this theory should reuse the assumptions used in the compositional security proofs of each component. Liang et al. [2014] developed a compositional refinement theory able to make use of general assumptions made by each component, which are guaranteed by the others. It remains to be seen whether *this* form of assume-guarantee reasoning [Jones 1981] can be reconciled with that supported by our theory from Section 3, inherited from Mantel et al. [2011].

Security Proofs of Compiler-Produced Implementations.

Combining the hypothetical compiler-produced proofs of source code security and implementation correctness, to prove that the compiled system is information flow secure, requires the security properties to be preserved by refinement. Many traditional ones are not, a result known as the *refinement paradox* [Jacob 1988].

Because our theory of Section 3 assumes a deterministic programming language, the refinement paradox should not trouble individual components. However, the implicit model of the scheduler in that theory, which allows any component to be scheduled at any time, is nondeterministic. In practice, the system scheduler implements some refinement of this behaviour. Like the original of Mantel et al. [2011], our global security property sys-secure is *not* preserved by refinement. Thus a system might be judged secure by sys-secure but when it is executed on a particular scheduler it may in fact be insecure.

This problem has been studied heavily, with one popular technique to address it being to define security properties that are *scheduler independent* [Sabelfeld and Sands 2000], meaning that they are preserved by any scheduler from a particular class. Sudbrock [2013] extended the theory of Mantel et al. [2011] to cover certain schedulers, so ours should be able to be extended likewise. Furthermore, by building on formally verified kernels like seL4 [Klein et al. 2014] we can also prove that *actual* schedulers on which a high-assurance language would be deployed meet the assumptions of the scheduler independence notion.

Deployment on a Verified Kernel.

Finally, the aforementioned theories should be able to be combined with those of a verified kernel, like seL4, beyond just reasoning about the scheduler. Specifically, the implementation correctness proofs will make certain assumptions about the underlying kernel on which the compiled code runs, including about the behaviour of system calls invoked by the compiled code, and that each component is correctly isolated from the others[2]. Both can be discharged by deploying on a verified kernel like seL4. seL4's functional correctness proofs [Klein et al. 2009] give system calls a precise, yet manageable, semantics; its security theorems, which cover both data integrity [Sewell et al. 2011] and confidentially [Murray et al. 2013], are ideal for discharging isolation assumptions.

A high-assurance information-flow-secure programming language with a self-certifying compiler, combined with a verified kernel like seL4, offers the possibility of unprecedented security assurance, leaving nowhere for vulnerabilities to hide: not in the application, nor in its compiled code, nor in the kernel on which it runs.

Acknowledgements

Thanks to June Andronick and the anonymous reviewers for their feedback on earlier drafts of this paper.
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

Andrew W. Appel and Daniel C. Wang. JVM TCB: Measurements of the trusted computing base of Java virtual machines. Technical Report TR-647-02, Princeton University, 2002.

Niklas Broberg, Bart van Delft, and David Sands. Paragon for practical programming with information-flow control. In *APLAS*, volume 8301 of *LNCS*, pages 217–232, 2013.

[2] Isolation provided by the underlying kernel allows static Asm-NoReadOrWrite assumptions to be trivially met.

Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *21st SOSP*, pages 31–44, 2007.

Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *S&P*, pages 354–368, 2008.

Dorothy. E. Denning. A lattice model of secure information flow. *CACM*, 19:236–242, 1976.

Joseph Goguen and José Meseguer. Security policies and security models. In *S&P*, pages 11–20, 1982.

Sylvia Grewe, Heiko Mantel, and Daniel Schoepe. A formalisation of assumptions and guarantees for compositional noninterference. *Archive of Formal Proofs*, 2014. http://afp.sourceforge.net/entries/SIFUM_Type_Systems.shtml.

Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: tracking information flow in JavaScript and its APIs. In *ACM Symp. Appl. Comput.*, pages 1663–1671, 2014.

Jeremy Jacob. Security specifications. In *S&P*, pages 14–23, 1988.

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

Gabi Keller, Toby Murray, Sidney Amani, Liam O'Connor-Davis, Zilin Chen, Leonid Ryzhyk, Gerwin Klein, and Gernot Heiser. File systems deserve verification too! In *PLOS*, pages 1–7, 2013.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, et al. seL4: Formal verification of an OS kernel. In *SOSP*, 2009.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *Trans. Comp. Syst.*, 32(1):2:1–2:70, 2014.

Xavier Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.

Hongjin Liang, Xinyu Feng, and Ming Fu. Rely-guarantee-based simulation for compositional verification of concurrent program transformations. *Trans. Progr. Lang. & Syst.*, 36(1):3:1–3:55, 2014.

Luísa Lourenço and Luís Caires. Dependent information flow types. In *POPL*, pages 317–328, 2015.

Heiko Mantel, David Sands, and Henning Sudbrock. Assumptions and guarantees for compositional noninterference. In *CSF*, pages 218–232, 2011.

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Noninterference for operating system kernels. In *CPP*, pages 126–142, 2012.

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *S&P*, pages 415–429, 2013.

Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.

Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. 2002.

Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *J. Selected Areas Comm.*, 21(1):5–19, 2003.

Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *CSFW*, pages 200–214, 2000.

Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In *Interactive Theorem Proving (ITP)*, pages 325–340, 2011.

Henning Sudbrock. *Compositional and Scheduler-Independent Information Flow Security*. PhD thesis, TU Darmstadt, 2013.

David von Oheimb. Information flow control revisited: Noninfluence = noninterference + nonleakage. In *9th ESORICS*, volume 3193 of *LNCS*, pages 225–243, 2004.

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