Security Policies for Downgrading

Stephen Chong
Department of Computer Science
Cornell University
schong@cs.cornell.edu

Andrew C. Myers
Department of Computer Science
Cornell University
andru@cs.cornell.edu

ABSTRACT

A long-standing problem in information security is how to specify and enforce expressive security policies that control information flow while also permitting information release (i.e., declassification) where appropriate. This paper presents security policies for downgrading and a security type system that incorporates them, allowing secure downgrading of information through an explicit declassification operation. Examples are given showing that the downgrading policy language captures useful aspects of designer intent. These policies are connected to a semantic security condition that generalizes noninterference, and the type system is shown to enforce this security condition.

Categories and Subject Descriptors: K.6.5 [Management of Computing and Information Systems]: Security and Protection

General Terms: Security, Languages

Keywords: Information flow, noninterference, downgrading, declassification, security policies.

1. INTRODUCTION

Control of information flow is an unavoidable aspect of enforcing security properties such as confidentiality and integrity. A long-standing problem is how to specify and enforce expressive policies for how information may flow. This problem is difficult because real-world applications release information as part of their intended function. This paper presents a framework for security policies that can express the intentional downgrading of information, and a security type system that incorporates and enforces these policies.

Strong information security properties (which we refer to broadly as *noninterference* [6]) specify an absence of information flow. As a result, they are too rigid to serve as a useful description of the security of realistic applications. Noninterference does not provide a way to distinguish between programs that release information as intended and programs that can leak information because of either programming error or vulnerability to attack.

Noninterference does have the attractive quality that it can be enforced by static analysis, an idea explored by work (e.g., [26, 8, 23, 29, 9, 20, 2, 30]) that augments programming-language types with security *labels* that capture information flow restrictions. An interesting question, therefore, is whether useful security properties can be defined for programs that intentionally release information. Type systems and program analyses have been defined that not only attempt to control information flow but also support *downgrading*, an escape hatch that enables information release by explicitly relaxing the security labels of data [17, 13, 31]. While downgrading seems to be usable in a principled way, it is unclear what security guarantees hold once it is used.

We call downgrading of confidentiality labels *declassification*. When a program declassifies information properly, there is some reason why it is acceptable for information to be released. Consider a database system storing confidential salary information. While individual salaries cannot be securely released, it may be acceptable to release an average of all salaries because the *amount* of information leaked about any individual salary is small. Another example is an online system for ordering software. In this case, the information being purchased (that is, the software) is initially confidential but should be released if the customer has paid for it.

As these examples suggest, the reasons for releasing information are diverse, often complex, yet crucial to security. We therefore propose a security policy framework that supports application-specific reasoning about downgrading in programs. A *declassification policy* specifies a sequence of security levels through which a given data item may move, where each step in the sequence is annotated with a condition that must be satisfied in order to perform the downgrading. This kind of policy captures simple temporal properties and can integrate external logics (such as access control) for reasoning about downgrading steps. We focus on security policies for declassification, but the same approach applies to other forms of downgrading, such as for integrity (e.g., [17, 31]).

This new framework has some important and novel properties:

- It is equipped with a semantic security condition that generalizes noninterference (and reduces to it in the absence of information release). This condition regulates the *end-to-end* behavior of a system rather than merely restricting individual downgrading operations as in intransitive noninterference [22, 18, 21, 11].
- To accommodate domain-specific reasoning about secure information release, the framework is parameterized with respect to a separate program analysis.
- It is possible to automatically and soundly check whether a system enforces the security policies for declassification, under the assumption that the application-specific analyses used to instantiate the framework can also be automatically and soundly checked.
- The policies are simple and intuitive.

CCS'04, October 25-29, 2004, Washington, DC, USA.

This paper extends a line of work that attempts to control when and what downgrading is allowed. The new policy framework integrates work on selective declassification, tied to an underlying access control mechanism, [14, 13, 19], with work on intransitive noninterference [22, 18, 21, 11]. Other ways to control information release include quantifying and limiting information flow (e.g., [12, 7, 25, 4, 10]) and ensuring that information release decisions are trustworthy [28, 16].

The remainder of the paper is organized as follows. Section 2 presents two examples that motivate the paper. The language of security policies for declassification is given in Section 3. Section 4 defines a programming language λ_{declass} that incorporates security policies for declassification. Section 5 presents a semantic security property of all well-typed λ_{declass} programs. The motivating examples are revisited in Section 6 using λ_{declass}. Section 7 offers some possible extensions, and related work is discussed in more detail in Section 8. Section 9 concludes. The appendix proves that the type system of λ_{declass} soundly enforces our semantic security condition.

2. MOTIVATING EXAMPLES

The goal of the security mechanism is to provide assurance to programmers that their programs release information in accordance with the overall system security policy. An underlying assumption is that the program executes as written: an attacker may be able to change inputs to the system but cannot change the code itself. Thus, the threat to security arises primarily from programming errors that are not detected by static analysis of information flow, but might lead to secrecy or privacy violations when the system is used.

This section contains two examples in which data is declassified. In the first example, a password checking system, data is susceptible to laundering: information that should remain secret may be declassified. In the second example, a sealed auction, the correct data is declassified, but possibly at the wrong time. In both of these examples, the security labels for the data are not expressive enough to capture the behavior that the overall system security policies require. In each system, programming errors might cause security violations even though the program passes the static information-flow analysis. It is this problem that we wish to address.

2.1 Password checking

Consider a system where the user must correctly enter a password before gaining access. Assuming the user can observe whether access has been granted, the system must release some information about the password: if the system grants the user access, then the system has released the information that the password is the same as the user's input; if the system denies access, then the user knows the password and the input differ. Furthermore, suppose the system contains some secret data that should not be released to the user; such data could be the passwords of other users, audit logs, or similar data. The system must ensure that when it releases information about the password, it does not also release any information about the secret data.

The following pseudocode shows an abstraction of such a system, where *pwd* is the password, *secret* is the secret data, and *guess* is the user's input. We assume for simplicity that the secret and the password are integers. The pseudocode uses security types, where types are labeled with security levels from some security lattice; the type system ensures that values labeled with a high security level do not depend on values labeled with low security levels. A program that type-checks in such a system can be shown to satisfy noninterference. We assume that the security level H (for "high security") is associated with secret data (including the password), and the security level L (for "low security") is associated with non-secret data.

```
1  int_H secret := ...;
2  int_H pwd := ...;
3  int_H guess := getUserInput();
4  boolean_H test := (guess=pwd);
5  boolean_L result := declassify(test, H ⇝ L);
   ...
```

Note that the user's guess is compared to the password (line 4), and the comparison is explicitly declassified from security level H to security level L (line 5). In a security-typed language setting, declassifying a value produces a copy of the value with a different security label.

Although this program is arguably secure, some similar programs are clearly not. For example, suppose the assignment $pwd := secret$ is inserted between lines 2 and 3. Since both *pwd* and *secret* have the security level H, such an assignment is legitimate. However, following the declassification at line 5, the user has gained some information about the secret data, namely, whether *secret* is identical to the user's guess. Such information release is known as *laundering*: existing declassifications are used to improperly declassify data. This example is susceptible to laundering because the security lattice is not expressive enough to distinguish the security policy for *pwd* from that of *secret*, even though the two items of data are to be used in different ways.

A common workaround to this lack of precision in a security lattice is to refine the security levels. For example, the security level H could be divided into H_1 for the password, and H_2 for the secret information that should not be released. However, this approach is unsatisfactory: the intrinsic meaning of the security levels may be lost. If two items of data should be treated the same, except that one item may be declassified while the other may not, then creating new security levels in an ad-hoc manner loses this connection between the data items. In addition, in larger systems, where there are many security levels through which data may be declassified, the ad-hoc creation of new security levels may become unmanageable.

2.2 Sealed auction

In a sealed auction, each bidder submits a single secret bid in a sealed envelope. Once all bids are submitted, the envelopes are opened and the bids compared; the winner is the highest bidder. A key security property is that no bidder knows any of the other bids until after all bids have been submitted.

The following security-typed pseudocode shows an abstraction of a sealed auction protocol with two bidders, Alice and Bob. The security lattice has a level A for data that only Alice can read, a security level B for data that only Bob can read, and a security level pub ("public") for data that both Alice and Bob can read.

```
1  int_A aliceBid := ...;
2  int_B bobBid := ...;
3  int_pub aliceOpenBid := declassify(aliceBid, A ⇝ pub);
4  int_pub bobOpenBid := declassify(bobBid, B ⇝ pub);
5  /* determine winner */
   ...
```

Note that Alice and Bob first submit their sealed bids (lines 1–2), and only then are the bids declassified (lines 3–4) and the winner of the auction is determined.

It is possible that the sealed auction protocol is implemented incorrectly (maybe through malice or programmer error), and one of the bids is declassified before both bids are submitted; this would

allow the other bidder to take advantage of incorrectly released information. For example, in the modified code below, Alice's bid is declassified too early, and Bob takes advantage of this to always win the auction with the lowest possible winning bid.

```
1  int_A aliceBid := ...;
2  int_pub aliceOpenBid := declassify(aliceBid, A ⇝ pub);
3  int_B bobBid := aliceOpenBid + 1;
4  int_pub bobOpenBid := declassify(bobBid, B ⇝ pub);
5  /* determine winner */
   ...
```

Alice may mistakenly decide that the system is secure, because she notes that her bid is initially secret and is later declassified; she may be unaware that the system declassifies her bid inappropriately.

Unlike in the password-checking example, the correct data (Alice's bid) is declassified but at the wrong time. Even though the system correctly enforces the specified security policies, it has a vulnerability because the security lattice cannot express when data may be declassified.

3. POLICIES FOR DECLASSIFICATION

In this section we present *declassification policies* which can specify how data should be used prior to declassification, under what conditions declassification is permitted, and how data should be treated after declassification. Declassification policies provide sufficient expressiveness to avoid the vulnerabilities of the examples of Section 2. Declassification policies are defined independently of any mechanism for enforcing them. In Section 4 we present a language that enforces declassification policies through a type system, but other enforcement mechanisms are possible, including run-time checking.

3.1 Policies

We assume there is some existing security lattice $\mathcal{L}$, such as the decentralized label model [15]. The elements of $\mathcal{L}$ are used in the specification of declassification policies, as follows.

If a security policy $\ell \overset{c}{\rightsquigarrow} p$ is enforced on some data, then the data must be used according to security level $\ell \in \mathcal{L}$, and may be declassified provided condition c is true at the time of declassification; after declassification, the security policy p is enforced on the declassified data.

Security policies for declassification:

$\ell \in \mathcal{L}$	Security levels from security lattice $\mathcal{L}$
$p ::=$	Security policies
$\quad \ell \overset{c}{\rightsquigarrow} p$	Declassification policy
$\quad \underline{\ell}$	Security level policy
$c ::=$	Conditions
$\quad d$	Primitive conditions
$\quad \mathbf{f}$	False
$\quad \mathbf{t}$	True
$\quad c \wedge c$	Conjunction
$\quad \neg c$	Negation

Conditions are used to express when it is appropriate to declassify data. Primitive conditions are assumed to have a truth value that may change during the execution of a program. As discussed in Section 3.2, primitive conditions are in general application-specific, and thus security policies for declassification are parameterized on the choice of primitive conditions. A condition $c_1 \wedge c_2$ is true at a given time only if both c_1 and c_2 are true at that time. A condition $\neg c$ is true at a given time only if the condition c is not true at that time. The condition $\mathbf{f}$ is never true, and the condition $\mathbf{t}$ is always true. When reasoning about conditions, we assume that the axioms and rules of classical propositional logic hold, and thus the language of conditions is equivalent to classical propositional logic. We abbreviate $\neg(\neg c \wedge \neg c')$ as the disjunction $c \vee c'$, and abbreviate $\neg c \vee c'$ as the implication $c \Rightarrow c'$.

For a given security level $\ell \in \mathcal{L}$, the policy $\underline{\ell}$ is used for data that may always be declassified to security level ℓ. The policy $\underline{\ell}$ is recursively equivalent to the policy $\ell \overset{\mathbf{t}}{\rightsquigarrow} \underline{\ell}$. The equivalence makes sense because declassification to the same security level is harmless and should always be permitted. Note that a syntactic distinction is made between security levels and declassification policies: ℓ is a security level, but $\underline{\ell}$ is a security policy for declassification.

We can define an ordering $\leq$ on policies, where $p \leq p'$ if policy p' is at least as restrictive as policy p. Intuitively, given two policies $p = \ell \overset{c}{\rightsquigarrow} p_0$ and $p' = \ell' \overset{c'}{\rightsquigarrow} p'_0$, the policy p' is at least as restrictive as p if security level ℓ' is at least as restrictive as security level ℓ, policy p'_0 is at least as restrictive as policy p_0, and whenever data associated with p' can be declassified to p'_0, data associated with p can be declassified to p_0. This last requirement, that data associated with p can be declassified whenever data associated with p' can, means that the condition c' should imply the condition c, that is, $c' \Rightarrow c$ should be true according to the axioms and rules of classical propositional logic and the semantics of the primitive conditions. The ordering $\leq$ is defined as the least relation consistent with the following rules and axioms. We use $\sqsubseteq_{\mathcal{L}}$ to denote the lattice ordering of the lattice $\mathcal{L}$.

Ordering $\leq$ for policies:

$$\frac{\ell \sqsubseteq_{\mathcal{L}} \ell' \quad p \leq p' \quad c' \Rightarrow c}{\ell \overset{c}{\rightsquigarrow} p \leq \ell' \overset{c'}{\rightsquigarrow} p'} \qquad \frac{\ell \sqsubseteq_{\mathcal{L}} \ell'}{\underline{\ell} \leq \underline{\ell'}} \qquad \frac{}{\underline{\ell} \leq \ell \overset{\mathbf{t}}{\rightsquigarrow} \underline{\ell}} \qquad \frac{}{\ell \overset{\mathbf{t}}{\rightsquigarrow} \underline{\ell} \leq \underline{\ell}}$$

The relation $\leq$ is not a partial order, as it is not antisymmetric: if we have two distinct conditions c and c' such that $c \Rightarrow c'$ and $c' \Rightarrow c$, then for any security level ℓ and policy p, we have $\ell \overset{c}{\rightsquigarrow} p \leq \ell \overset{c'}{\rightsquigarrow} p$ and $\ell \overset{c'}{\rightsquigarrow} p \leq \ell \overset{c}{\rightsquigarrow} p$; also, for any security level ℓ, we have $\underline{\ell} \leq \ell \overset{\mathbf{t}}{\rightsquigarrow} \underline{\ell}$ and $\ell \overset{\mathbf{t}}{\rightsquigarrow} \underline{\ell} \leq \underline{\ell}$.

However, if we define the equivalence relation $\equiv$ over policies such that $p \equiv p'$ if and only if $p \leq p'$ and $p' \leq p$, and identify all policies that are equivalent according to $\equiv$, then the relation $\leq$ is well-defined on the resulting structure, and moreover forms a join semi-lattice, with join operation $\sqcup$. The top element of the join semi-lattice, which we denote $\top$, is equivalent to the infinitely long policy $\top_{\mathcal{L}} \overset{\mathbf{f}}{\rightsquigarrow} \top_{\mathcal{L}} \overset{\mathbf{f}}{\rightsquigarrow} \cdots$, where $\top_{\mathcal{L}}$ is the top element of the lattice $\mathcal{L}$; the top policy $\top$ is thus recursively equivalent to $\top_{\mathcal{L}} \overset{\mathbf{f}}{\rightsquigarrow} \top$. Note that $\top$ is not equivalent to $\underline{\top_{\mathcal{L}}}$. The bottom element of the join semi-lattice is $\underline{\bot_{\mathcal{L}}}$, where $\bot_{\mathcal{L}}$ is the bottom element of $\mathcal{L}$.

The join operation $\sqcup$ arises naturally from $\leq$, but for clarity we present it here, using $\sqcup_{\mathcal{L}}$ to denote the join operation of $\mathcal{L}$.

Join operation $\sqcup$ for policies:

$$\frac{\ell \sqcup_{\mathcal{L}} \ell' = \ell'' \quad p \sqcup p' = p'' \quad c'' = c \wedge c'}{(\ell \overset{c}{\rightsquigarrow} p) \sqcup (\ell' \overset{c'}{\rightsquigarrow} p') = (\ell'' \overset{c''}{\rightsquigarrow} p'')} \qquad \frac{\ell \sqcup_{\mathcal{L}} \ell' = \ell''}{\underline{\ell} \sqcup \underline{\ell'} = \underline{\ell''}}$$

In the remainder of the paper, we assume that all policies are identified up to the equivalence relation $\equiv$, and thus the set of policies under the partial order $\leq$ forms a join semi-lattice.

There is no restriction on the security levels that may appear in a declassification policy. In particular, if a policy allows information to be declassified from ℓ to ℓ' (e.g., $\ell \overset{c}{\rightsquigarrow} \underline{\ell'}$), then $\ell' \sqsubseteq_{\mathcal{L}} \ell$, $\ell \sqsubseteq_{\mathcal{L}} \ell'$, and ℓ incomparable to ℓ' are all acceptable.

3.2 Conditions

The conditions of the declassification policies are used to express when data may be declassified. Since reasons for information release are highly varied and depend on the application domain, it is important that the choice and definition of conditions can also be specific to the application domain. Towards this end, we place very few restrictions on conditions.

The framework is abstract with respect to the choice of primitive conditions, and in general they may be specific to the application domain in which declassification policies are used. The framework is also abstract with respect to the semantics of primitive conditions; they could, for example, be defined in terms of application-domain semantics, or as properties of program states. However, as will be seen in Section 4, when using declassification policies in a language setting, it is useful to be able to express the semantics of conditions in terms of the language semantics.

We require that logical implication and conjunction are defined for conditions, but this is simply to allow the ordering $\leq$ and the join operation $\sqcup$ to be defined. In this presentation we assume the rules and axioms of classical propositional logic for conditions, but other logics are possible, as is briefly discussed in Section 7.

Declassification policies as presented here are thus very general. When used in a particular application domain, the declassification policy framework would be instantiated through the choice and meaning of the primitive conditions (and possibly a different logic over primitive conditions); the results of this paper are applicable to any such instantiation.

Declassification policies, like intransitive noninterference, can express what information may flow between security levels. However, the use of conditions in declassification policies adds temporal structure that intransitive noninterference is unable to express. Declassification policies can express *when* information is permitted to flow between security levels.

3.3 Motivating examples

Declassification policies have enough expressive power to avoid the vulnerabilities of the examples in Section 2.

In the password-checking example, we can use the security policy $\underline{H}$ for the secret data *secret*, indicating that the secret data must be used according to the security level H, and can never be declassified below that security level. By contrast, the security policy for the password is $H \overset{cert}{\rightsquigarrow} \underline{L}$, indicating that it must be used according to the security level H, but that it may be declassified to the policy $\underline{L}$ provided the condition $cert$ is true at the time of declassification, where $cert$ is a primitive condition that is true only when certified and trusted code is executing. Since the security policy for the secret data prevents any declassification below level H, a system that enforces these security policies will never declassify the secret data to level L, thus preventing laundering. Declassification policies are expressive enough to differentiate between data that can be declassified, and data that cannot.

The sealed auction example was susceptible to an early declassification of Alice's sealed bid. Let the primitive condition $bids$ be true if and only if both Alice and Bob have submitted their sealed bids. A suitable security policy for Alice's bid is then $A \overset{bids}{\rightsquigarrow} \underline{pub}$, and similarly $B \overset{bids}{\rightsquigarrow} \underline{pub}$ is suitable for Bob's bid. With these security policies, Alice's bid cannot be declassified before Bob has submitted his bid, since the condition $bids$ would not be true at this time. Declassification policies can express under what conditions data should be declassified.

4. A LANGUAGE FOR DECLASSIFICATION

In this section we present λ_{declass}, a language based on the typed lambda calculus in which declassification policies are used as labels for types, and the type system ensures that the security policies associated with data are enforced. The language is imperative, has a store, and has an explicit declassification operator. The following subsections present the syntax of λ_{declass}, the small-step operational semantics, and the static semantics. To increase generality, the type system of λ_{declass} is parameterized on a *static condition analysis*—a static analysis that can determine at which program points the conditions are true. The choice of primitive conditions will influence which static analyses can be used to instantiate the type system.

4.1 Syntax

The syntax of λ_{declass} is given below. Metavariables x and y range over variable names, and m and n range over memory locations.

A type β_p in λ_{declass} consists of a base type β annotated with a declassification policy p. Memory locations are tagged with the type of the value contained in the location; a location m^τ can only store values of type τ. For a function $\lambda x : \tau.[p]\ e$ of base type $\tau \xrightarrow{p} \tau'$, the security policy p is a lower bound on the memory effects of the function; that is, executing the function body e will only allocate or update memory locations $m^{\beta_{p'}}$ where $p \leq p'$.

λ_{declass} contains a declassification operator: *declassify*$(v, \ell \rightsquigarrow \ell')$ is used to declassify the value v from security level ℓ to security level ℓ'. Declassifying a value produces a copy of the value with a different security label. The type system ensures that declassification can only occur if appropriate conditions are true at the time of declassification.

Syntax:

$v ::=$	Values
$\quad x$	Variables
$\quad n$	Integers
$\quad ()$	Unit
$\quad \lambda x : \tau.[p]\ e$	Abstraction
$\quad m^\tau$	Memory locations
$e ::=$	Expressions
$\quad v$	Values
$\quad e\ e$	Application
$\quad \mathbf{ref}^\tau\ e$	Allocation
$\quad !e$	Dereference
$\quad e := e$	Assignment
$\quad e; e$	Sequence
$\quad$ *declassify*$(v, \ell \rightsquigarrow \ell')$	Declassification
$\beta ::=$	Base types
$\quad$ **int**	Integers
$\quad$ **unit**	Unit
$\quad \tau \xrightarrow{p} \tau'$	Functions
$\quad \tau$ **ref**	References
$\tau ::=$	Security types
$\quad \beta_p$	Base types with policies

4.2 Operational semantics

The small-step operational semantics of λ_{declass} are presented in Figure 1. A memory M is a finite map from typed locations to closed values; thus, $M(m^\tau)$ is the value stored in the typed memory location m^τ. A configuration is a pair of an expression e and a memory M, written $(\!|e, M|\!)$. A small evaluation step is a transition from one configuration $(\!|e, M|\!)$ to another configuration $(\!|e', M'|\!)$, written $(\!|e, M|\!) \longrightarrow (\!|e', M'|\!)$.

It is necessary to restrict the form of configurations $(\!|e, M|\!)$ to avoid using undefined memory locations. A memory M is well-formed if every address m appears at most once in $\mathrm{dom}(M)$, and for every $m^\tau \in \mathrm{dom}(M)$, we have $\mathrm{loc}(M(m^\tau)) \subseteq \mathrm{dom}(M)$, where $\mathrm{loc}(e)$ is the set of typed locations occurring in the expression e. A configuration $(\!|e, M|\!)$ is well-formed if M is well-formed, $\mathrm{loc}(e) \subseteq \mathrm{dom}(M)$, and e contains no free variables.

We use $e[v/x]$ to denote the capture-avoiding substitution of value v for variable x in the expression e. If M is a memory, $M[m^\tau \mapsto v]$ denotes the memory obtained by mapping location m^τ to v in M.

The operational semantics ensure that λ_{declass} is a deterministic call-by-value language. Note that the *declassify* expression has no computational effect: when a value is declassified from one security level to another, the value is not modified in any way.

4.3 Static semantics

The subtyping relationship $<:$ uses the partial order $\leq$ on policies, and plays an important role in the enforcement of information flow security. We write $\tau <: \tau'$ if security type τ is a subtype of security type τ'. We overload the $<:$ symbol by defining a subtyping relationship on base types, writing $\beta <: \beta'$ if base type β is a subtype of base type β'.

Subtyping:

$$\frac{\beta <: \beta' \quad p \leq p'}{\beta_p <: \beta'_{p'}} \qquad \frac{}{\beta <: \beta} \qquad \frac{\beta <: \beta' \quad \beta' <: \beta''}{\beta <: \beta''} \qquad \frac{p' \leq p \quad \tau'_1 <: \tau_1 \quad \tau_2 <: \tau'_2}{\tau_1 \xrightarrow{p} \tau_2 <: \tau'_1 \xrightarrow{p'} \tau'_2}$$

Note that if β_p is a subtype of ${\beta'}_{p'}$, then the security policy p' is at least as restrictive as the security policy p. This is consistent with the usual notion of subtyping [24].

The typing judgment $pc, \Gamma \vdash e : \tau$ means that expression e has type τ under variable context Γ and the policy pc is a lower bound on memory effects of e. The judgment $\vdash M$ means that memory M is well-typed, that is, every location of M stores an appropriately typed value. A configuration $(\!|e, M|\!)$ is well-typed if M is well-typed and $pc, \emptyset \vdash e : \tau$ for some pc and τ. Typing judgments on values, expressions and memories are defined in Figure 2.

Note that rules (T-ALLOC) and (T-ASSIGN) ensure that pc is a lower bound on the memory effects of a well-typed expression. For a function with base type $\tau \xrightarrow{p} \tau'$, p is a lower bound on the memory effects of the function, and an upper bound on the pc context of the caller, as evidenced by the rules (T-ABS) and (T-APP).

The rule for declassification (T-DECLASS) has an interesting and important side-condition: it must be statically provable that the condition for declassification is true whenever the declassification occurs. The side-condition ensures that the policy for declassification is respected.

The rule (T-DECLASS) is parameterized by a static condition analysis that can determine at which program points conditions are true. The choice of an appropriate static condition analysis depends on the relation of the semantics of conditions to the program semantics, and thus on the choice of primitive conditions (and the logic over primitive conditions); a suitable analysis might be a type system, dataflow analysis, or an abstract interpretation, for example. Parameterizing the type system on a static condition analysis means that the results of this paper hold regardless of the choice of primitive conditions. Section 6 presents the motivating examples in terms of λ_{declass}, and gives some instantiations of the type system with specific static condition analyses.

λ_{declass} has the standard properties of progress and type preservation (which are not presented or proven here) and is thus sound.

5. SECURITY PROPERTIES

This section presents a semantic security property possessed by all well-typed programs in λ_{declass}. This property is reminiscent of (and generalizes) noninterference, but permits information to be released through permitted declassifications.

5.1 Equivalence relation $\approx_\ell$

It is first necessary to define what the attacker at security level ℓ can observe; this is accomplished by defining an equivalence relation $\approx_\ell$ over typed values. Intuitively, given two values v and v' of the same base type β, $v \approx_\ell v'$ if the two values are indistinguishable to an attacker who can only make observations at level ℓ or lower. Note that a value with security policy $p = \ell' \overset{c'}{\rightsquigarrow} p'$ is observable at level ℓ if and only if $\ell' \sqsubseteq_{\mathcal{L}} \ell$, or equivalently, $p \leq \ell \overset{\mathbf{f}}{\rightsquigarrow} \top$. Figure 3 presents the definition of the equivalence relation.

The relation $\approx_\ell$ is a conservative, syntactic notion of equivalence. In particular, if two functions are observable to an attacker who can only make observations at security level ℓ or lower, then the two functions must be syntactically identical for the attacker to regard them as equivalent; the attacker can distinguish two functions that are contextually equivalent, but syntactically different.

We extend $\approx_\ell$ to an equivalence relation over memories using bijections between the memory locations at or below security level ℓ. For a memory M, we define $\mathrm{dom}(M|\ell)$ to be the locations of M at or below security level ℓ:

$$\mathrm{dom}(M|\ell) \triangleq \{m^{\beta_p} \mid m^{\beta_p} \in \mathrm{dom}(M) \wedge p \leq \ell \overset{\mathbf{f}}{\rightsquigarrow} \top\}$$

If M_1 and M_2 are memories, then $M_1 \approx_\ell M_2$ if there exists a bijection f from $\mathrm{dom}(M_1|\ell)$ to $\mathrm{dom}(M_2|\ell)$ such that f preserves types, and moreover if f maps location m^τ in M_1 to n^τ in M_2, then $M_1(m^\tau) \approx_\ell M_2(n^\tau)$ if τ is not a reference type, and $f(M_1(m^\tau)) = M_2(n^\tau)$ if τ is a reference type. Intuitively, $M_1 \approx_\ell M_2$ if the structure of the observable portion of M_1 (that is, $\mathrm{dom}(M_1|\ell)$) is the same as the structure of the observable portion of M_2, and in addition the non-reference values in the observable portions are indistinguishable. We write $[M]_{\approx_\ell}$ to denote the equivalence class of memory M under the equivalence relation $\approx_\ell$.

5.2 Evaluations

Given a well-formed configuration $(\!|e_0, M_0|\!)$, an *evaluation* of $(\!|e_0, M_0|\!)$ is a sequence of configurations $(\!|e_0, M_0|\!) \ldots (\!|e_n, M_n|\!)$, such that $(\!|e_{i-1}, M_{i-1}|\!) \longrightarrow (\!|e_i, M_i|\!)$ for all $i \in 1..n$. An evaluation is *partial* if e_n is not a value.

An evaluation is c-free if no step in the evaluation reduces a term of the form $\mathit{declassify}(v, \ell \rightsquigarrow \ell')$ with the condition c true, for any v, ℓ and ℓ'. Note that all evaluations are **f**-free, since **f** is never true.

For a given sequence of conditions $c_1 \ldots c_k$, an evaluation E is $c_1 \ldots c_k$-free if E is c_1-free, or if for all evaluations E_1 and E_2 such that $E = E_1 E_2$ and E_1 is not c_1-free, then E_2 is $c_2 \ldots c_k$-free. Intuitively, an evaluation is $c_1 \ldots c_k$-free if it does not contain k declassifications where c_i was true at the ith declassification.

Given a (partial) evaluation $E = (\!|e_0, M_0|\!) \ldots (\!|e_n, M_n|\!)$, we define the memory trace of E to be the sequence $M_0 \ldots M_n$, and the $\approx_\ell$-memory trace of E, denoted $[E]_{\approx_\ell}$, to be the sequence $[M_0]_{\approx_\ell} \ldots [M_n]_{\approx_\ell}$.

5.3 Noninterference until conditions $c_1, \ldots, c_k$

Noninterference [6] is the security property that secret inputs do not affect non-secret outputs. The precise definitions of inputs and outputs lead to slightly different definitions of noninterference. We assume a λ_{declass} program has a single integer input in the form of a free variable, and the output of a program evaluation is its memory trace. Noninterference for λ_{declass} is then defined as follows.

(β-REDUCTION)
$$\frac{}{(\!|(\lambda x : \tau[p].\ e)\ v, M|\!) \longrightarrow (\!|e[v/x], M|\!)}$$

(ALLOCATION)
$$\frac{}{(\!|\mathbf{ref}^{\tau}\ v, M|\!) \longrightarrow (\!|m^{\tau}, M[m^{\tau} \mapsto v]|\!)}\ m^{\tau} \text{ fresh}$$

(DEREF)
$$\frac{}{(\!|!m^{\tau}, M|\!) \longrightarrow (\!|M(m^{\tau}), M|\!)}$$

(ASSIGN)
$$\frac{}{(\!|m^{\tau} := v, M|\!) \longrightarrow (\!|(), M[m^{\tau} \mapsto v]|\!)}$$

(SEQ)
$$\frac{}{(\!|(); e, M|\!) \longrightarrow (\!|e, M|\!)}$$

(DECLASS)
$$\frac{}{(\!|\mathit{declassify}(v, \ell \rightsquigarrow \ell'), M|\!) \longrightarrow (\!|v, M|\!)}$$

(CONTEXT)
$$\frac{(\!|e, M|\!) \longrightarrow (\!|e', M'|\!)}{(\!|E[e], M|\!) \longrightarrow (\!|E[e'], M'|\!)}$$

Contexts:
$E ::= [\cdot]\ e \mid v\ [\cdot] \mid \mathbf{ref}^{\tau}\ [\cdot] \mid ![\cdot] \mid [\cdot] := e \mid v := [\cdot] \mid [\cdot]; e$

Figure 1: Operational Semantics

(T-VAR)
$$\frac{\Gamma(x) = \tau}{pc, \Gamma \vdash x : \tau}$$

(T-INT)
$$\frac{}{pc, \Gamma \vdash n : \mathbf{int}_p}$$

(T-UNIT)
$$\frac{}{pc, \Gamma \vdash () : \mathbf{unit}_p}$$

(T-LOC)
$$\frac{}{pc, \Gamma \vdash m^{\tau} : \tau\ \mathbf{ref}_p}$$

(T-SUB)
$$\frac{pc, \Gamma \vdash e : \tau \quad \tau <: \tau'}{pc, \Gamma \vdash e : \tau'}$$

(T-DEREF)
$$\frac{pc, \Gamma \vdash e : \beta_p\ \mathbf{ref}_{p'}}{pc, \Gamma \vdash !e : \beta_{p \sqcup p'}}$$

(T-SEQ)
$$\frac{pc, \Gamma \vdash e_1 : \mathbf{unit}_p \quad pc, \Gamma \vdash e_2 : \tau}{pc, \Gamma \vdash e_1; e_2 : \tau}$$

(T-ABS)
$$\frac{p, \Gamma[x \mapsto \tau] \vdash e : \tau'}{pc, \Gamma \vdash \lambda x : \tau.[p]\ e : (\tau \xrightarrow{p} \tau')_{p'}}$$

(T-APP)
$$\frac{pc, \Gamma \vdash e_1 : (\tau \xrightarrow{pc'} \beta'_{p'})_p \quad pc, \Gamma \vdash e_2 : \tau \quad pc \leq pc'}{pc, \Gamma \vdash e_1\ e_2 : \beta'_{p' \sqcup p}}$$

(T-ALLOC)
$$\frac{pc, \Gamma \vdash e : \beta_p \quad pc \leq p}{pc, \Gamma \vdash \mathbf{ref}^{\beta_p}\ e : (\beta_p\ \mathbf{ref})_{p'}}$$

(T-ASSIGN)
$$\frac{pc, \Gamma \vdash e_1 : \beta_p\ \mathbf{ref}_{p'} \quad pc, \Gamma \vdash e_2 : \beta_p \quad pc \sqcup p' \leq p}{pc, \Gamma \vdash e_1 := e_2 : \mathbf{unit}_{p'}}$$

(T-MEM)
$$\frac{\forall m^{\tau} \in \mathrm{dom}(M).\ \top, \emptyset \vdash M(m^{\tau}) : \tau}{\vdash M}$$

(T-DECLASS)
$$\frac{pc, \Gamma \vdash v : \beta_{\ell \overset{c}{\rightsquigarrow} \ell'}\ \overset{c'}{\rightsquigarrow} p}{pc, \Gamma \vdash \mathit{declassify}(v, \ell \rightsquigarrow \ell') : \beta_{\ell'}\ \overset{c'}{\rightsquigarrow} p}$$

condition c is true whenever $\mathit{declassify}(v, \ell \rightsquigarrow \ell')$ is reduced with rule (DECLASS)

Figure 2: Typing judgments

DEFINITION 5.1.: *Let e be an expression with free variable x : $\mathbf{int}_p$, and let $\ell \in \mathcal{L}$ be a security level. Expression e is* noninterfering for x at level ℓ *if for all integer values v_1 and v_2, and all memories M such that $(\!|e[v_1/x], M|\!)$ and $(\!|e[v_2/x], M|\!)$ are well-formed and well-typed, and for any evaluations E_1 of $(\!|e[v_1/x], M|\!)$ and E_2 of $(\!|e[v_2/x], M|\!)$, then $[E_i]_{\approx_\ell}$ is a prefix up to stuttering[1] of $[E_j]_{\approx_\ell}$, where $\{i, j\} = \{1, 2\}$.* ∎

Intuitively, a program is noninterfering for x at level ℓ if an attacker who can observe all memory locations at level ℓ or lower cannot distinguish any two executions of the program with different inputs x. The requirement that one $\approx_\ell$-memory trace is a prefix of the other up to stuttering implies that the attacker can observe changes to memory but is unable to measure the time between changes. Thus our definition of noninterference is *timing-insensitive*. It is also *termination-insensitive*: the attacker is unable to distinguish a program that has terminated from a program that is still running but hasn't modified any memory since the last change.

In the presence of declassification, a given program e may not be noninterfering, even if e is well-typed. For example, the program $\mathbf{ref}^{\mathbf{int}_L}$ ($\mathit{declassify}(x, H \rightsquigarrow L)$) is well-typed (where x is of type $\mathbf{int}_{H \overset{t}{\rightsquigarrow} L}$ and $H \not\sqsubseteq_{\mathcal{L}} L$), but it is not noninterfering for x at level L, since an attacker can distinguish evaluations with different values of x. Therefore a generalization of noninterference is needed.

The property *noninterference until conditions $c_1, \ldots, c_k$* is based on the intuition that an attacker cannot have made any observations about the input if the following holds:

- The attacker can only make observations about the input after some data has been declassified to some level ℓ or lower;
- the data is only declassified to level ℓ or lower after a sequence of k declassifications, with condition c_i true at the ith declassification; and
- that sequence of declassifications has not yet occurred.

DEFINITION 5.2.: *Let e be an expression with free variable x : $\mathbf{int}_{\ell_1 \overset{c_1}{\rightsquigarrow} \cdots \overset{c_{k-1}}{\rightsquigarrow} \ell_k \overset{c_k}{\rightsquigarrow} p}$, and let $\ell \in \mathcal{L}$ be a security level. Expression e is* noninterfering for x at level ℓ until conditions $c_1, \ldots, c_k$ *if for all integer values v_1 and v_2, and all memories M such that $(\!|e[v_1/x], M|\!)$ and $(\!|e[v_2/x], M|\!)$ are well-formed and well-typed, and for any $c_1 \ldots c_k$-free evaluations E_1 of $(\!|e[v_1/x], M|\!)$ and E_2 of $(\!|e[v_2/x], M|\!)$ then $[E_i]_{\approx_\ell}$ is a prefix up to stuttering of $[E_j]_{\approx_\ell}$, where $\{i, j\} = \{1, 2\}$.* ∎

Evaluations E_1 and E_2 are $c_1 \ldots c_k$-free, so the sequence of k declassifications, with c_i true at the ith declassification, cannot

[1] Sequence s_1 is a prefix up to stuttering of sequence s_2 if s'_1 is a prefix of s'_2, where s'_i is the result of removing all consecutively repeated elements from s_i. For example, the sequence $aabacc$ is a prefix up to stuttering of $abbbacd$, since $abac$ is a prefix of $abacd$.

$$\frac{}{() : \mathbf{unit}_{p_1} \approx_\ell () : \mathbf{unit}_{p_2}} \qquad \frac{p_1 \le \ell \overset{\mathbf{f}}{\rightsquigarrow} \top \quad p_2 \le \ell \overset{\mathbf{f}}{\rightsquigarrow} \top}{n : \mathbf{int}_{p_1} \approx_\ell n : \mathbf{int}_{p_2}} \qquad \frac{p_1 \not\le \ell \overset{\mathbf{f}}{\rightsquigarrow} \top \quad p_2 \not\le \ell \overset{\mathbf{f}}{\rightsquigarrow} \top}{n_1 : \mathbf{int}_{p_1} \approx_\ell n_2 : \mathbf{int}_{p_2}} \qquad \frac{p_1 \le \ell \overset{\mathbf{f}}{\rightsquigarrow} \top \quad p_2 \le \ell \overset{\mathbf{f}}{\rightsquigarrow} \top}{m^\tau : \tau\ \mathbf{ref}_{p_1} \approx_\ell m^\tau : \tau\ \mathbf{ref}_{p_2}}$$

$$\frac{p_1 \not\le \ell \overset{\mathbf{f}}{\rightsquigarrow} \top \quad p_2 \not\le \ell \overset{\mathbf{f}}{\rightsquigarrow} \top}{m_1^\tau : \tau\ \mathbf{ref}_{p_1} \approx_\ell m_2^\tau : \tau\ \mathbf{ref}_{p_2}} \qquad \frac{p_1 \le \ell \overset{\mathbf{f}}{\rightsquigarrow} \top \quad p_2 \le \ell \overset{\mathbf{f}}{\rightsquigarrow} \top}{(\lambda x : \tau.[p]\ e) : (\tau \xrightarrow{p} \tau')_{p_1} \approx_\ell (\lambda x : \tau.[p]\ e) : (\tau \xrightarrow{p} \tau')_{p_2}} \qquad \frac{p_1 \not\le \ell \overset{\mathbf{f}}{\rightsquigarrow} \top \quad p_2 \not\le \ell \overset{\mathbf{f}}{\rightsquigarrow} \top}{(\lambda x : \tau.[p_1']\ e_1) : (\tau_1 \xrightarrow{p_1'} \tau_1')_{p_1} \approx_\ell (\lambda y : \tau.[p_2']\ e_2) : (\tau_2 \xrightarrow{p_2'} \tau_2')_{p_2}}$$

Figure 3: Equivalence relation $\approx_\ell$

have occured in either evaluation. If the expression e is noninterfering for x at level ℓ until conditions $c_1, \ldots, c_k$, then an attacker that can only observe memory at level ℓ or lower is thus unable to distingush evaluation E_1 from evaluation E_2.

If the input is never declassified below a certain level, noninterference until conditions $c_1, \ldots, c_k$ entails noninterference at that level, as stated by the following theorem:

THEOREM 5.3.**:** *Let e be an expression with free variable x : $\mathbf{int}_{\ell_1} \overset{c_1}{\rightsquigarrow} \cdots \overset{c_{k-1}}{\rightsquigarrow} \ell_k \overset{\mathbf{f}}{\rightsquigarrow} p$, and let $\ell \in \mathcal{L}$ be a security level such that e is noninterfering for x at level ℓ until conditions $c_1, \ldots, c_{k-1}, \mathbf{f}$. Then e is noninterfering for x at level ℓ.*

The type system of λ_{declass} ensures that well-typed programs are noninterfering until conditions $c_1, \ldots, c_k$, for all appropriate sequences of conditions $c_1 \ldots c_k$, and appropriate security levels.

THEOREM 5.4.**:** *Let e be an expression such that $pc, \emptyset[x \mapsto \mathbf{int}_{\ell_1} \overset{c_1}{\rightsquigarrow} \cdots \overset{c_{k-1}}{\rightsquigarrow} \ell_k \overset{c_k}{\rightsquigarrow} p] \vdash e : \tau$ for some security policy pc and type τ, and let $\ell \in \mathcal{L}$ be a security level such that $\ell_i \not\sqsubseteq_{\mathcal{L}} \ell$ for all $i \in 1..k$. Then e is noninterfering for x at level ℓ until conditions $c_1, \ldots, c_k$.*

Note that Theorem 5.4 ensures that a well-typed e is noninterfering until conditions $c_1, \ldots, c_k$ only at security levels ℓ such that $\ell_i \not\sqsubseteq_{\mathcal{L}} \ell$ for all $i \in 1..k$; the intuition is that if $\ell_i \sqsubseteq_{\mathcal{L}} \ell$ for some $i \in 1..k$, then the attacker may observe information about the input before all k declassifications have been performed.

The proof of Theorem 5.4 uses Pottier and Simonet's proof technique [20], which extends the language to allow a single expression to represent two program executions that differ only in their secret inputs; the proof of Theorem 5.4 is reduced to a type preservation proof in the extended language. We present the key points of the proof here, and refer the reader to Appendix A for more details.

The language $\lambda^2_{\text{declass}}$ extends the language λ_{declass} with a bracket construct $\langle e_1 \mid e_2 \rangle$. The pair $\langle e_1 \mid e_2 \rangle$ represents two different expressions, e_1 and e_2, that may arise during two different executions of a program. Thus, the two λ_{declass} expressions $e[v_1/x]$ and $e[v_2/x]$ can be represented by the single expression $e[\langle v_1 \mid v_2 \rangle / x]$ in $\lambda^2_{\text{declass}}$. A bracket expression can appear arbitrarily deep within an expression, but cannot be nested within another bracket expression. Using $\lambda^2_{\text{declass}}$, Theorem 5.4 can be proved in three steps:

1. Prove that $\lambda^2_{\text{declass}}$ adequately represents the execution of two λ_{declass} expressions. Pottier and Simonet's definition of adequacy is not suited for our purposes; their extended language is required to represent two base language evaluations only if both base language evaluations terminate. Since we need to reason about partial evaluations, our definition of adequacy requires that given two λ_{declass} evaluations, there is a $\lambda^2_{\text{declass}}$ evaluation that fully represents at least one of them.
2. Prove type preservation for $\lambda^2_{\text{declass}}$. A key point of the type system of $\lambda^2_{\text{declass}}$ is that if the bracket expression $\langle e_1 \mid e_2 \rangle$ has type $\beta_{p'}$, then both e_1 and e_2 have type $\beta_{p'}$, and moreover, for some fixed policy $\ell_1 \overset{c_1}{\rightsquigarrow} \cdots \overset{c_{k-1}}{\rightsquigarrow} \ell_k \overset{c_k}{\rightsquigarrow} p$ there is some $j \in 1..k$ such that $\ell_j \overset{c_j}{\rightsquigarrow} \cdots \overset{c_{k-1}}{\rightsquigarrow} \ell_k \overset{c_k}{\rightsquigarrow} p \le p'$. This ensures that for any level $\ell \in \mathcal{L}$ such that $\ell_i \not\sqsubseteq_{\mathcal{L}} \ell$ for all $i \in 1..k$, no bracket expression is observable at level ℓ.
3. Prove that type preservation for $\lambda^2_{\text{declass}}$ entails Theorem 5.4. Let e be an expression that is well-typed under the variable context containing the single variable $x : \mathbf{int}_{\ell_1} \overset{c_1}{\rightsquigarrow} \cdots \overset{c_{k-1}}{\rightsquigarrow} \ell_k \overset{c_k}{\rightsquigarrow} p$. Let $\ell \in \mathcal{L}$ be a security level such that $\ell_i \not\sqsubseteq_{\mathcal{L}} \ell$ for all $i \in 1..k$. Let v_1 and v_2 be two integers. Let M_0 be a memory such that $(\!|e[v_1/x], M_0|\!)$ and $(\!|e[v_2/x], M_0|\!)$ are well-formed and well-typed. The $\lambda^2_{\text{declass}}$ configuration $(\!|e[\langle v_1 \mid v_2 \rangle / x], M_0|\!)$ represents the two λ_{declass} configurations $(\!|e[v_1/x], M_0|\!)$ and $(\!|e[v_2/x], M_0|\!)$. Consider $(\!|e[\langle v_1 \mid v_2 \rangle / x], M_0|\!) \ldots (\!|e_n, M_n|\!)$, a (partial) $c_1 \ldots c_k$-free evaluation of $(\!|e[\langle v_1 \mid v_2 \rangle / x], M_0|\!)$ (which represents two $c_1 \ldots c_k$-free evaluations, one of the configuration $(\!|e[v_1/x], M_0|\!)$ and one of $(\!|e[v_2/x], M_0|\!)$). Type preservation for $\lambda^2_{\text{declass}}$ ensures that a value bound to any memory location in M_n that is observable at security level ℓ contains no bracket constructs. Thus, the two sequences of memories from the two λ_{declass} evaluations are $\approx_\ell$ equivalent up to stuttering, showing that e is noninterfering for x at level ℓ until conditions $c_1, \ldots, c_k$, as required.

6. MOTIVATING EXAMPLES REVISITED

In this section we reconsider the examples of Section 2 in terms of the language λ_{declass}, and see how the type system of the language avoids the vulnerabilities mentioned.

6.1 Password checking

Consider the abstraction of the password checking system, written in λ_{declass}, assuming the addition of booleans, equality test $==$ for integers, and the use of **let** $x{:}\tau = e$ **in** e' as syntactic sugar for $(\lambda x : \tau.[p]\ e')\ e$, for some appropriate security policy p.

```
1  let secret:int_L = ... in
2  let pwd:(int_{H ⇝^cert L}) ref_pub = ... in
3  let guess:int_{H ⇝^cert L} = ... in
4  let test:boolean_{H ⇝^cert L} = (guess == !pwd) in
5  let result:boolean_L = declassify(test, H ⇝ L) in
   ...
```

Recall that the type system of λ_{declass} requires a static condition analysis to determine that the appropriate conditions for declassification are true whenever declassification occurs. The primitive condition $cert$ is true only when certified and trusted code is running. Thus, a suitable static condition analysis can simply check if the *declassify* expression occurs in certified and trusted code. Since

line 5 is certified and trusted code (as it is part of the trusted password checking module), $cert$ is true when the *declassify* expression of line 5 is reduced, and so the program is well-typed.

The type system for λ_{declass} prevents the value stored in the memory location *pwd* from depending on the value of *secret*. If the assignment *pwd* := *secret* were inserted between lines 2 and 3, type checking would reject the program, as $\mathbf{int}_{\underline{H}}$, the type of *secret*, is not a subtype of $\mathbf{int}_{H \overset{cert}{\rightsquigarrow} \underline{L}}$, the type of the password.

More generally, no laundering of data can be well-typed in λ_{declass}, as declassification policies can express what declassifications are permitted, and the type system ensures these security policies are enforced. This includes laundering through *implicit flows* of information, where the control structure of a program is used as an information channel.

6.2 Sealed auctions

Consider the abstraction of the sealed auction protocol written in λ_{declass}, where the primitive condition $bids$ is true only when both Alice and Bob's bids have both been submitted.

```
1  let aliceBid:int_{A ~>^{bids} pub} = ... in
2  let bobBid:int_{B ~>^{bids} pub} = ... in
3  let aliceOpenBid:int_{pub} = declassify(aliceBid, A ~> pub) in
4  let bobOpenBid:int_{pub} = declassify(bobBid, B ~> pub) in
5  /* determine winner */
   ...
```

Unlike the password checking example, the connection between the semantics of the primitive conditions and the program semantics is non-trivial, and thus a more complex static condition analysis is required. However, for the abstraction of the auction protocol given above, a relatively simple static condition analysis is suitable.

Note that Alice's bid is the value of the variable *aliceBid* and Bob's bid is the value of the variable *bobBid*, and that these values are immutable after their initialization. Thus, the primitive condition $bids$ is true at a given program point only if both *aliceBid* and *bobBid* are in the variable context at that program point. At lines 3 and 4, where Alice and Bob's bids are declassified, both variables are in the context, and so the condition $bids$ is true at these program points, and the program is well-typed.

If the program were modified by swapping lines 2 and 3 (as in Section 2), so that Alice's bid were declassified before Bob's bid is submitted, then $bids$ would not be true at the early declassification of Alice's bid: the variable *bobBid* would not be in the context. The modified program would thus fail to type-check. The type system of λ_{declass} ensures that the condition for declassification must be true at the time of declassification, and so prevents the inappropriate early declassification of Alice's bid.

A fuller and more realistic implementation of the sealed auction protocol (having, for example, multiple auctions, a statically unknown number of bidders, and functions for program modularization), would require a correspondingly more complex static condition analysis. Note, however, that the declassification policies for bids would remain the same; what changes is the relationship of the primitive condition semantics to the program semantics.

7. EXTENSIONS

There are several possible extensions to the declassification policies which increase policy expressiveness and allow them to capture more precisely the intended security behavior of systems.

Other logics for conditions: Classical propositional logic is used to reason about the conditions for declassification. For a given application domain, other logics may be better suited to express when declassification is permitted; possibilities include first-order logic, temporal logic, epistemic logic and linear logic. The logic must define an appropriate form of conjunction and implication for conditions to support the ordering $\leq$ and join operation $\sqcup$.

(a) Trees (b) Arbitrary Rooted Directed Graphs

Figure 4: Examples of extended security policies for operations other than declassify.

For example, consider the sealed auction of Section 2. Temporal logic can express a suitable condition for when Alice and Bob's bids can be declassified: $\mathbf{P}bidsClosed$, where $bidsClosed$ is true when the auction bidding stops, and the formula $\mathbf{P}bidsClosed$ is true when the auction bidding stopped sometime in the past. Standard model checking techniques provide a sound static condition analysis that can determine if temporal logic formulas are always true at the time of declassification, that is, if for all possible execution paths to the declassification, the formula is true.

The use of more expressive logics may make the comparison of two security policies (and thus the type checking of λ_{declass} programs) undecidable. In addition, proving that a condition is always true at the time of declassification, as required by the rule (T-DECLASS), may become more complex.

Operations other than declassify: Extending the security policies to other operations than declassification increases their expressiveness. Data labeled with an extended policy $\ell \overset{c:\, op}{\rightsquigarrow} p$ must be treated at security level ℓ, the operator op may be applied to the data provided condition c is true, and the result of the operation is labeled with security policy p. The security policy $\ell \overset{c}{\rightsquigarrow} p$ is equivalent to the extended security policy $\ell \overset{c:\, declassify}{\rightsquigarrow} p$, where *declassify* is the declassify operator.

With multiple operators, there is no reason to restrict the number of possible operations on data; extended security policies could form trees, such as in Figure 4(a) or even arbitrary rooted directed graphs, as in Figure 4(b).

The password example of Section 6.1 could benefit from the more expressive security policies. The security policy for the password is $H \overset{cert}{\rightsquigarrow} \underline{L}$, which permits the password in its entirety to be declassified to security level L. The introduction of an equality test operator *eq* would allow the policy to be amended to $H \overset{cert:\, eq}{\rightsquigarrow} \underline{L}$, which expresses the desirable restriction [25] that only the result of an equality test of the password can be declassified.[2]

Further investigation of extended security policies may show them to be expressive enough to allow the clean treatment of encryption and decryption as primitive operations. Suppose *enc* is a primitive encryption operator. Then the security policy $H \overset{c:\, enc}{\rightsquigarrow} \underline{L}$ would allow the encryption of secret (security level H) data, and allow the resulting encrypted data to be non-secret (security level L); the policy would also remove the possibility of accidental declassification of unencrypted data, which the policy $H \overset{c}{\rightsquigarrow} \underline{L}$ might permit.

Dynamic security policies: The λ_{declass} typing rule for declas-

[2]It is also necessary to ensure that results of computations involving the password cannot be declassified with equality tests [25], which would allow the entire password to be leaked in just k equality tests, where k is the password size in bits. This can be accomplished by adding an operator *arith* which represents all arithmetic operations; the password's security policy wouldn't allow any arithmetic operations on the password, removing this possible channel.

sification (T-DECLASS) requires the static proof that condition c is true when the declassification of data labeled $\ell \overset{c}{\rightsquigarrow} p$ occurs. This static proof burden could be removed and a more dynamic approach to declassification policies taken. Instead, a declassification could succeed or fail at run time, depending on whether the condition is true or false at the time of declassification. However, since the run-time behavior of the program would now depend on the security policies of data, a new information channel is introduced, and care must be taken that the desired information flow properties hold regardless of the success or failure of declassifications. Other work on dynamic security policies [32] addresses these issues.

8. RELATED WORK

Intransitive noninterference [22, 18, 21] is an information flow property based on noninterference that was introduced to describe the behavior of systems that need to declassify information. Intransitive noninterference is an intensional property, where in each step of computation, information only flows between security levels (or *domains*) according to some (possibly intransitive) relation.

Declassification policies are compatible with intransitive noninterference. A security policy $\ell \overset{c}{\rightsquigarrow} \ell' \overset{c'}{\rightsquigarrow} p$ can be viewed as a declaration that information may flow from security level ℓ to security level ℓ'. Alternatively, a security policy $\ell \overset{c}{\rightsquigarrow} \ell' \overset{c'}{\rightsquigarrow} p$ may only be allowed if information is permitted to flow from ℓ to ℓ', according to some (externally declared) relation. In fact, declassification policies extend intransitive noninterference with temporal properties: in each computation step, information flows between levels only if that flow is permitted *and* appropriate conditions are true for that computation step. This additional expressiveness permits security policies that prevent inappropriate declassifications, such as in the sealed auction example, which intransitive noninterference by itself is unable to accomplish.

Intransitive noninterference specifies security policies on security domains, that is, how information may flow between domains; by contrast, declassification policies associate security policies with data. For example, assuming $H, L \in \mathcal{L}$ are domains such that $H \not\sqsubseteq_{\mathcal{L}} L$, data labeled with the security policy $H \overset{c}{\rightsquigarrow} \underline{L}$ is in the same domain H as data labeled with the security policy $\underline{H}$; however, the former may be declassified to the domain L, while the latter will never be declassified to the domain L. Thus, declassification policies provide additional precision, ensuring that data labeled $\underline{H}$ will never be declassified to the domain L, despite the fact that some information is permitted to flow from domain H to L. This additional precision could alternatively be seen as a refinement of security domains: the policy $\ell \overset{c}{\rightsquigarrow} p$ represents the subdomain of ℓ that may be declassified (when c is true) to the subdomain represented by p. Viewed in this light, security policies for declassification provide a structured and intuitive method of refining security domains, based on what declassifications may be performed in the future.

Recent work by Mantel and Sands [11] places intransitive noninterference in a language setting, providing a bisimulation-based security condition for multi-threaded programs that controls where information can be declassified, and a type system for a language that enforces this condition.

Robust declassification [28] is a desirable security property for systems that perform declassification. In brief, a system is robust if an active attacker (one who can observe and modify the behavior of the system) cannot learn more about the system (including secret inputs) than could a passive attacker (one who can observe but not modify the behavior of a system). In a language based setting, this means that the decision to declassify data must be trusted [27, 16]; this is equivalent to the following condition being true at every declassification: *control flow and data at this program point is trusted*. By making this a primitive condition and conjoining it to all declassification conditions in the security policies, robust declassification can be expressed in λ_{declass} using an appropriate static condition analysis. The requirement that the decision to declassify data must be trusted is alluded to in the password checking example in Section 6.1, through use of the primitive condition $cert$, which is true only if certified and trusted code is executing.

Selective declassification [19] was introduced as part of the decentralized label model [14, 15], and requires the owners of data to authorize all declassifications of that data; which owners are required to give their authorization for a given declassification depends on what security levels the data is being declassified from and to. Pottier and Conchon [19] present selective declassification as a combination of information flow and access control, where a number of declassification operations are locked at appropriate levels of authority; access control allows only suitably authorized principals to unlock the declassification operations, and only unlocked declassification operations can declassify information. Selective declassification, like robust declassification, attempts to prevent inappropriate declassifications by requiring a certain condition to be true when declassification occurs: *all required owners of the data have authorized the declassification*. Like robust declassification, such a condition can be incorporated into declassification policies.

Banerjee and Naumann give a type system for a Java-like language that uses access control to mediate information release [3]. The type system allows a dependency to be introduced between dynamically enabled access control permissions in the Java model and the security level of a method result. The security property enforced is, however, noninterference.

Ferrari et al. [5] use a form of dynamically-checked declassification in an object-oriented system through *waivers* to strict information flow. Waivers are applied dynamically and can mention specific data objects, thus providing fine-grain control over when information is declassified. With suitable extensions to the declassification policies (specifically dynamic testing of conditions and possibly a more expressive logic for conditions than classical propositional logic), we believe that waivers can be represented using declassification policies.

The language λ_{declass} is a *security-typed language* (e.g., [26, 24, 8, 13, 1, 2, 20]), in which types of program variables are annotated with security policies. The type systems of such language enforce security properties, typically noninterference.

Other approaches, such as *quantitative information flow* (e.g., [12, 10, 4]) and *relative secrecy* [25] seek to measure or bound the amount of information that is declassified. This work is largely orthogonal to declassification policies, which, in the context of this paper, are concerned only with possibilistic security assurances. However, the conditions for declassifications could perhaps be useful in specifying or bounding channel capacities.

9. CONCLUSION

We have presented an expressive framework for declassification security policies, and incorporated them in a security type system. A security policy for declassification describes a sequence of security levels through which a labeled data value may be declassified if associated conditions are met. Security policies for declassification are defined independently of any mechanism for enforcing them; the security type system presented here is one such mechanism.

In the language setting of a security type system, these declassification policies are connected to a semantic security condition that generalizes noninterference to allow information release only if the given conditions are satisfied. For generality, we have parameterized the declassification policy framework on the choice of

conditions, and correspondingly parameterized the type system on a static condition analysis that connects the semantics of the conditions to the language semantics. Our experience in applying these policies to various small programs, some of which are given here, suggests that the policy language is intuitive and usefully restricts program behavior. Thus, the analysis embodied in the type system helps to avoid writing insecure programs in the presence of downgrading. We have also identified a number of possible extensions that may lead to future work.

Acknowledgments

Thanks to Andrei Sabelfeld and Steve Zdancewic for suggestions about declassification policies, and Michael Clarkson, Nate Nystrom, Riccardo Pucella, Lantian Zheng, and the anonymous reviewers for providing helpful feedback. Dave Sands also offered some useful insights on intransitive noninterference.

This work was supported by the Department of the Navy, Office of Naval Research, under ONR Grant N00014-01-1-0968. Any opinions, findings, conclusions, or recommendations contained in this material are those of the authors and do not necessarily reflect views of the Office of Naval Research. This work was also supported by the National Science Foundation under grants 0208642 and 0133302, and by an Alfred P. Sloan Research Fellowship.

10. REFERENCES

[1] J. Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 40–53, Boston, MA, Jan. 2000.

[2] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*, June 2002.

[3] A. Banerjee and D. A. Naumann. Using access control for secure information flow in a java-like language. In *Proc. 16th IEEE Computer Security Foundations Workshop*, pages 155–169, June 2003.

[4] A. Di Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In *Proc. 15th IEEE Computer Security Foundations Workshop*, pages 1–15, June 2002.

[5] E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 130–140, Oakland, CA, USA, May 1997.

[6] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20, Apr. 1982.

[7] J. W. Gray, III. Towards a mathematical foundation for information flow security. In *Proc. IEEE Symposium on Security and Privacy*, pages 21–34, 1991.

[8] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, California, Jan. 1998.

[9] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 81–92. ACM Press, Jan. 2002.

[10] G. Lowe. Quantifying information flow. In *Proc. 15th IEEE Computer Security Foundations Workshop*, June 2002.

[11] H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. Unpublished draft, 2003.

[12] J. K. Millen. Covert channel capacity. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, 1987.

[13] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, Jan. 1999.

[14] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, Saint-Malo, France, 1997.

[15] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symposium on Security and Privacy*, pages 186–197, Oakland, CA, USA, May 1998.

[16] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proc. 17th IEEE Computer Security Foundations Workshop*, June 2004. to appear.

[17] J. Palsberg and P. Ørbæk. Trust in the λ-calculus. In *Proc. 2nd International Symposium on Static Analysis*, number 983 in Lecture Notes in Computer Science, pages 314–329. Springer, Sept. 1995.

[18] S. Pinsky. Absorbing covers and intransitive non-interference. In *Proc. IEEE Symposium on Security and Privacy*, pages 102–113, 1995.

[19] F. Pottier and S. Conchon. Information flow inference for free. In *Proc. 5nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 46–57, 2000.

[20] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 319–330, 2002.

[21] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proc. 12th IEEE Computer Security Foundations Workshop*, 1999.

[22] J. Rushby. Noninterference, transitivity and channel-control security policies. Technical Report CSL-92-02, SRI, Dec. 1992.

[23] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. 13th IEEE Computer Security Foundations Workshop*, pages 200–214. IEEE Computer Society Press, July 2000.

[24] D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of the 7th International Joint Conference on the Theory and Practice of Software Development*, pages 607–621, 1997.

[25] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 268–276, Boston, MA, Jan. 2000.

[26] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[27] S. Zdancewic. A type system for robust declassification. In *Proceedings of the Nineteenth Conference on the Mathematical Foundations of Programming Semantics*, Electronic Notes in Theoretical Computer Science, Mar. 2003.

[28] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Nova Scotia, Canada, June 2001.

[29] S. Zdancewic and A. C. Myers. Secure information flow and CPS. In *Proc. 10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 46–61, 2001.

[30] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. 16th IEEE Computer Security Foundations Workshop*, pages 29–43, Pacific Grove, California, June 2003.

[31] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proc. 18th ACM Symp. on Operating System Principles (SOSP)*, pages 1–14, Banff, Canada, Oct. 2001.

[32] L. Zheng and A. C. Myers. Dynamic security labels and noninterference. Technical Report 2004–1924, Cornell University Computing and Information Science, 2004.

APPENDIX

A. PROOF OF THEOREM 5.4

In this appendix we present the syntax and semantics of the language $\lambda^2_{\text{declass}}$, show that it is adequate to represent the evaluation of two λ_{declass} expressions, and that type preservation holds. Finally, we prove that type preservation of $\lambda^2_{\text{declass}}$ implies Theorem 5.4.

A.1 Syntax and Semantics of $\lambda^2_{\text{declass}}$

The language $\lambda^2_{\text{declass}}$ extends the language λ_{declass} with a bracket construct $\langle e_1 \mid e_2 \rangle$. The pair $\langle e_1 \mid e_2 \rangle$ represents two different ex-

pressions, e_1 and e_2, that may arise in two different executions of a program. A bracket expression may not be nested inside a bracket expression, but can otherwise appear nested at arbitrary depth.

In addition to tracking subexpressions that may differ in different executions of a program, we also need to track how the memories may differ. Since we have dynamic allocation of memory locations, the language $\lambda^2_{\text{declass}}$ includes a special constant value **void**; if memory location m^τ is bound to a value $\langle \textbf{void} \mid v \rangle$ or $\langle v \mid \textbf{void} \rangle$, then m^τ is bound in only one of the two executions.

$\lambda^2_{\text{declass}}$ Syntax:

$v ::=$	values
...	λ_{declass} values
$\langle v \mid v \rangle$	Pair
void	Void
$e ::=$	expressions
...	λ_{declass} expressions
$\langle e \mid e \rangle$	Pair

Given an extended expression e, let the projections $\lfloor e \rfloor_1$ and $\lfloor e \rfloor_2$ represent the two λ_{declass} expressions that e encodes. The projection functions satisfy $\lfloor \langle e_1 \mid e_2 \rangle \rfloor_i = e_i$ and are homomorphisms on other expression forms. The projection functions are extended to memories in the following way: $\lfloor M \rfloor_i$ maps the memory location m^τ to $\lfloor M(m^\tau) \rfloor_i$ if and only if $\lfloor M(m^\tau) \rfloor_i$ is defined and isn't **void**. In addition, $\langle e_1 \mid e_2 \rangle[v/x]$, the capture-free substitution of v for x in $\langle e_1 \mid e_2 \rangle$, must use the corresponding projection of v in each branch: $\langle e_1 \mid e_2 \rangle[v/x] = \langle e_1[\lfloor v \rfloor_1/x] \mid e_2[\lfloor v \rfloor_2/x] \rangle$.

We extend configurations to triples $(\!| e, M |\!)_i$ for an index $i \in \{\bullet, 1, 2\}$. The index indicates if the expression e represents a pair of expressions ($\bullet$) or the left (1) or right (2) side of a pair of expressions. A configuration $(\!| e, M |\!)_i$ is well-formed if the following conditions are true: e does not contain **void**; if $i \in \{1, 2\}$ then e does not contain a bracket construct and $(\!| e, \lfloor M \rfloor_i |\!)$ is a well-formed λ_{declass} configuration; if $i = \bullet$ then $(\!| \lfloor e \rfloor_1, \lfloor M \rfloor_1 |\!)$ and $(\!| \lfloor e \rfloor_2, \lfloor M \rfloor_2 |\!)$ are well-formed λ_{declass} configurations.

The operational semantics of $\lambda^2_{\text{declass}}$ are presented in Figure 5. They are based on the semantics of λ_{declass}, and contain some new evaluation rules: (BRACKET), (LIFT-β), (LIFT-ASSIGN), (LIFT-DEREF), (BRACKET-SEQ). The rules (ALLOCATION), (BRACKET) and (DEREF) are modified to access the memory projection corresponding to index i, and the remaining λ_{declass} rules are adapted to $\lambda^2_{\text{declass}}$ by indexing each configuration with i. The evaluation contexts are the same in both languages.

The typing system for $\lambda^2_{\text{declass}}$ contains all typing rules of λ_{declass}, with the addition of two new rules, given below. For notational convenience, we define the $H \trianglelefteq p$ ("H protects p") as follows.

DEFINITION A.1. *If H is a policy $\ell_1 \overset{c_1}{\rightsquigarrow} \cdots \overset{c_{k-1}}{\rightsquigarrow} \ell_k \overset{c_k}{\rightsquigarrow} p$, and p' is an arbitrary policy, then we write $H \trianglelefteq p'$ to denote that there exists a $j \in 1..k$ such that $\ell_j \overset{c_j}{\rightsquigarrow} \cdots \overset{c_k}{\rightsquigarrow} p \leq p'$. Similarly, we write $H \sqcup p'' \trianglelefteq p'$ to denote that there exists a $j \in 1..k$ such that $(\ell_j \overset{c_j}{\rightsquigarrow} \cdots \overset{c_k}{\rightsquigarrow} p) \sqcup p'' \leq p'$. Finally, we write $H \not\trianglelefteq p$ if it is not the case that $H \trianglelefteq p$.* ∎

Typing judgments for $\lambda^2_{\text{declass}}$:

$$\frac{}{pc, \Gamma \vdash \textbf{void} : \tau} \text{(T-VOID)} \qquad \frac{H \sqcup pc \trianglelefteq pc' \quad H \trianglelefteq p \quad pc', \Gamma \vdash e_1 : \beta_p \quad pc', \Gamma \vdash e_2 : \beta_p}{pc, \Gamma \vdash \langle e_1 \mid e_2 \rangle : \beta_p} \text{(T-BRACKET)}$$

Note that the typing rule (T-BRACKET) is parameterized with a security policy H, and ensures that $H \sqcup pc$ protects the memory effects of both subexpressions, and also that H protects the security policy for the type of a bracket expression. This property will be key in the proof of type preservation.

A configuration $(\!| e, M |\!)_i$ is well-typed if M is well-typed and $pc, \emptyset \vdash e : \tau$ for some pc and τ.

For a sequence of conditions $c_1 \ldots c_k$, an evaluation $(\!| e_0, M_0 |\!)_\bullet \ldots (\!| e_n, M_n |\!)_\bullet$ in $\lambda^2_{\text{declass}}$ is $c_1 \ldots c_k$-free if both projections of the evaluation are $c_1 \ldots c_k$-free, i.e., if both evaluations $(\!| \lfloor e_0 \rfloor_1, \lfloor M_0 \rfloor_1 |\!) \ldots (\!| \lfloor e_n \rfloor_1, \lfloor M_n \rfloor_1 |\!)$ and $(\!| \lfloor e_0 \rfloor_2, \lfloor M_0 \rfloor_2 |\!) \ldots (\!| \lfloor e_n \rfloor_2, \lfloor M_n \rfloor_2 |\!)$ are $c_1 \ldots c_k$-free. Note that for the projections of the evaluations, for all $i \in 1..n$ and $j \in \{1, 2\}$ we have $(\!| \lfloor e_{i-1} \rfloor_j, \lfloor M_{i-1} \rfloor_j |\!) \longrightarrow^= (\!| \lfloor e_i \rfloor_j, \lfloor M_i \rfloor_j |\!)$, (where $\longrightarrow^=$ is the reflexive closure of $\longrightarrow$) instead of $(\!| \lfloor e_{i-1} \rfloor_j, \lfloor M_{i-1} \rfloor_j |\!) \longrightarrow (\!| \lfloor e_i \rfloor_j, \lfloor M_i \rfloor_j |\!)$; the notion of $c_1 \ldots c_k$-freeness can be extended for these evaluations easily.

A.2 Adequacy of $\lambda^2_{\text{declass}}$

The extended language $\lambda^2_{\text{declass}}$ is adequate for reasoning about the execution of two λ_{declass} expressions. We show that evaluation of a $\lambda^2_{\text{declass}}$ is both sound (a reduction of a bracket expression corresponds to a reduction of one of its projections) and complete, for a precise notion of "completeness" that is sufficient for our purposes.

LEMMA A.2. ***(Soundness):*** *If $(\!| e, M |\!)_\bullet \longrightarrow (\!| e', M' |\!)_\bullet$ then $(\!| \lfloor e \rfloor_i, \lfloor M \rfloor_i |\!) \longrightarrow^= (\!| \lfloor e' \rfloor_i, \lfloor M' \rfloor_i |\!)$ for $i \in \{1, 2\}$.*

Proof: By induction on the derivation $(\!| e, M |\!)_\bullet \longrightarrow (\!| e', M' |\!)_\bullet$. The only interesting case is (BRACKET), where we need to appeal to the fact that if $(\!| e, M |\!)_i \longrightarrow (\!| e', M' |\!)_i$, then $(\!| e, \lfloor M \rfloor_i |\!) \longrightarrow (\!| e', \lfloor M' \rfloor_i |\!)$, which follows from inspection of the rules (ALLOCATION), (BRACKET) and (DEREF). ∎

LEMMA A.3. ***(Stuck Configurations):*** *If $(\!| e, M |\!)_\bullet$ is stuck (i.e. it cannot be reduced, and e is not a value), then $(\!| \lfloor e \rfloor_i, \lfloor M \rfloor_i |\!)$ is stuck for some $i \in \{1, 2\}$.*

Proof: By induction on the structure of e. ∎

As mentioned in Section 5.3, Pottier and Simonet's definition of adequacy is not suitable for reasoning about partial evaluations. We define completeness such that if we have two λ_{declass} evaluations, then there is a $\lambda^2_{\text{declass}}$ evaluation that fully represents at least one of them; this is sufficient for us to later prove that an $\approx_\ell$-memory trace of one (partial) evaluation is a prefix (up to stuttering) of the $\approx_\ell$-memory trace of the other.

LEMMA A.4. ***(Completeness):*** *If $(\!| \lfloor e \rfloor_i, \lfloor M \rfloor_i |\!) \longrightarrow^* (\!| e'_i, M'_i |\!)$ for $i \in \{1, 2\}$, then there exists a configuration $(\!| e', M' |\!)_\bullet$ such that $(\!| e, M |\!)_\bullet \longrightarrow^* (\!| e', M' |\!)_\bullet$ and either $(\!| \lfloor e' \rfloor_1, \lfloor M' \rfloor_1 |\!) = (\!| e'_1, M'_1 |\!)$ or $(\!| \lfloor e' \rfloor_2, \lfloor M' \rfloor_2 |\!) = (\!| e'_2, M'_2 |\!)$.*

Proof: Let $(\!| e^i_0, M^i_0 |\!) \ldots (\!| e^i_{n_i}, M^i_{n_i} |\!)$ where we have $(\!| e^i_0, M^i_0 |\!) = (\!| \lfloor e \rfloor_i, \lfloor M \rfloor_i |\!)$ and $(\!| e^i_{n_i}, M^i_{n_i} |\!) = (\!| e'_i, M'_i |\!)$ for $i \in \{1, 2\}$, i.e., $(\!| \lfloor e \rfloor_i, \lfloor M \rfloor_i |\!) \longrightarrow^* (\!| e'_i, M'_i |\!)$. For evaluations $E = (\!| e_0, M_0 |\!)_\bullet \ldots (\!| e_n, M_n |\!)_\bullet$ define the function $f_i(E)$ to be the number of evaluation steps that reduced the ith projection: $f_i(E) = |\{k \mid 0 \leq k \leq n-1 \wedge \lfloor e_k \rfloor_i \neq \lfloor e_{k+1} \rfloor_i\}|$. Suppose that we have an evaluation E, starting from $(\!| e, M |\!)_\bullet$. Consider the function $g(E) = \min(n_1 - f_1(E), n_2 - f_2(E))$. Clearly if $g(E)$ is zero, then E is an evaluation from $(\!| e, M |\!)_\bullet$ to some configuration $(\!| e', M' |\!)_\bullet$ such that either $(\!| \lfloor e' \rfloor_1, \lfloor M' \rfloor_1 |\!) = (\!| e'_1, M'_1 |\!)$ or $(\!| \lfloor e' \rfloor_2, \lfloor M' \rfloor_2 |\!) = (\!| e'_2, M'_2 |\!)$, which suffices to prove the lemma. The proof constructs such an evaluation, by showing that if $g(E)$ is not zero, then by Lemma A.3

Operational semantics of $\lambda^2_{\text{declass}}$

(ALLOCATION)
$$\frac{}{(\!|\mathbf{ref}^\tau\ v, M|\!)_i \longrightarrow (\!|m^\tau, M[m^\tau \mapsto \text{new}_i(v)]|\!)_i}\quad m^\tau \text{ fresh}$$

(DEREF)
$$\frac{}{(\!|!m^\tau, M|\!)_i \longrightarrow (\!|\text{read}_i(M(m^\tau)), M|\!)_i}$$

(ASSIGN)
$$\frac{}{(\!|m^\tau := v, M|\!)_i \longrightarrow (\!|(), M[m^\tau \mapsto \text{update}_i(M(m^\tau), v)]|\!)_i}$$

(BRACKET)
$$\frac{(\!|e_i, M|\!)_i \longrightarrow (\!|e_i', M'|\!)_i \quad e_j' = e_j \quad \{i,j\} = \{1,2\}}{(\!|\langle e_1 \mid e_2\rangle, M|\!)_\bullet \longrightarrow (\!|\langle e_1' \mid e_2'\rangle, M'|\!)_\bullet}$$

(LIFT-DEREF)
$$\frac{}{(\!|!\langle v_1 \mid v_2\rangle, M|\!)_\bullet \longrightarrow (\!|\langle !v_1 \mid !v_2\rangle, M|\!)_\bullet}$$

(BRACKET-SEQ)
$$\frac{}{(\!|\langle () \mid ()\rangle; e, M|\!)_\bullet \longrightarrow (\!|e, M|\!)_\bullet}$$

(LIFT-β)
$$\frac{}{(\!|\langle v_1 \mid v_2\rangle\ v, M|\!)_\bullet \longrightarrow (\!|\langle v_1\ \lfloor v\rfloor_1 \mid v_2\ \lfloor v\rfloor_2\rangle, M|\!)_\bullet}$$

(LIFT-ASSIGN)
$$\frac{}{(\!|\langle v_1 \mid v_2\rangle := v, M|\!)_\bullet \longrightarrow (\!|\langle v_1 := \lfloor v\rfloor_1 \mid v_2 := \lfloor v\rfloor_2\rangle, M|\!)_\bullet}$$

Auxiliary Functions:

$\text{new}_\bullet(v) = v$	$\text{read}_\bullet(v) = v$	$\text{update}_\bullet(v, v') = v'$
$\text{new}_1(v) = \langle v \mid \mathbf{void}\rangle$	$\text{read}_1(v) = \lfloor v\rfloor_1$	$\text{update}_1(v, v') = \langle v' \mid \lfloor v\rfloor_2\rangle$
$\text{new}_2(v) = \langle \mathbf{void} \mid v\rangle$	$\text{read}_2(v) = \lfloor v\rfloor_2$	$\text{update}_2(v, v') = \langle \lfloor v\rfloor_1 \mid v'\rangle$

Figure 5: Operational Semantics of $\lambda^2_{\text{declass}}$

we can extend E using (BRACKET), to some evaluation E', such that $f_i(E') = f_i(E) + 1$ for some $i \in \{1, 2\}$, thus eventually constructing an evaluation E'' such that $g(E'')$ is zero. ∎

A.3 Type Preservation for $\lambda^2_{\text{declass}}$

The type preservation theorem for $\lambda^2_{\text{declass}}$ is nonstandard, and assumes that there is some distinguished configuration $(\!|e_0, M_0|\!)_\bullet$ that the computation started from, and that if we have a reduction $(\!|e, M|\!)_\bullet \longrightarrow (\!|e', M'|\!)_\bullet$, then there is an evaluation of $(\!|e_0, M_0|\!)_\bullet$ ending in the configuration $(\!|e, M|\!)_\bullet$. The type preservation theorem needs knowledge of the entire history of the computation, since the proof of Theorem 5.4 only requires that type preservation for $\lambda^2_{\text{declass}}$ holds for $c_1 \ldots c_k$-free evaluations.

THEOREM A.5. ***(Type Preservation):*** *Suppose $H = \ell_1 \overset{c_1}{\rightsquigarrow} \cdots \overset{c_{k-1}}{\rightsquigarrow} \ell_k \overset{c_k}{\rightsquigarrow} p_H$, $(\!|e, M|\!)_i \longrightarrow (\!|e', M'|\!)_i$, $(\!|e, M|\!)_i$ is well-formed and well-typed, $pc, \emptyset \vdash e : \tau$, $\vdash M$ and $i \in \{1, 2\}$ implies $H \trianglelefteq pc$, and $i = \bullet$ implies that the evaluation $(\!|e_0, M_0|\!)_\bullet \ldots (\!|e, M|\!)_\bullet (\!|e', M'|\!)_\bullet$ is $c_1 \ldots c_k$-free. Then $pc, \emptyset \vdash e' : \tau$, $\vdash M'$, dom$(M) \subseteq$ dom(M') and $(\!|e', M'|\!)_i$ is well-formed and well-typed.*

Proof: By induction on the derivation of $(\!|e, M|\!)_i \longrightarrow (\!|e', M'|\!)_i$. The most interesting case is declassification, where the expression e is *declassify*$(v, \ell \rightsquigarrow \ell')$ and e' is v and $M' = M$ and τ is $\beta_{\ell'} \overset{c'}{\rightsquigarrow} p$. By (T-DECLASS), $pc, \emptyset \vdash v : \beta_\ell \overset{c}{\rightsquigarrow} \ell' \overset{c'}{\rightsquigarrow} p$. Now we need to show that $pc, \emptyset \vdash v : \beta_{\ell'} \overset{c'}{\rightsquigarrow} p$. Consider the possible forms of v. Note that v cannot be a variable (since the context is empty). If v is an integer, void, unit, location or abstraction, then the appropriate typing rules for values ((T-INT), (T-VOID), (T-UNIT), (T-LOC) and (T-ABS)) allow a value of base type β to be given any policy, including $\ell' \overset{c'}{\rightsquigarrow} p$. If v is a bracket value $\langle v_1 \mid v_2\rangle$, then we know $H \trianglelefteq \ell \overset{c}{\rightsquigarrow} \ell' \overset{c'}{\rightsquigarrow} p$, that is, there exists some $j \in 1..k$ such that $\ell_j \overset{c_j}{\rightsquigarrow} \cdots \overset{c_{k-1}}{\rightsquigarrow} \ell_k \overset{c_k}{\rightsquigarrow} p_H \leq \ell \overset{c}{\rightsquigarrow} \ell' \overset{c'}{\rightsquigarrow} p$. Moreover, since v is a bracket value, $i = \bullet$, implying the evaluation is $c_1 \ldots c_k$-free; thus, we can show by induction on k that j must be less than k. Thus $\ell_{j+1} \overset{c_{j+1}}{\rightsquigarrow} \cdots \overset{c_{k-1}}{\rightsquigarrow} \ell_k \overset{c_k}{\rightsquigarrow} p_H \leq \ell' \overset{c'}{\rightsquigarrow} p$ and $j + 1$ is less than or equal to k, and so $H \trianglelefteq \ell' \overset{c'}{\rightsquigarrow} p$ as required. ∎

The type preservation of $\lambda^2_{\text{declass}}$ implies Theorem 5.4, which we are ready to prove, after one additional lemma.

LEMMA A.6.**:** *Let H be an arbitrary policy $\ell_1 \overset{c_1}{\rightsquigarrow} \cdots \overset{c_{k-1}}{\rightsquigarrow} \ell_k \overset{c_k}{\rightsquigarrow} p_H$. Let $m^{\beta p}$ be a memory location such that $H \not\trianglelefteq p$. Let M be a memory with $m^{\beta p} \in$ dom(M). Let $(\!|e, M|\!)_i \longrightarrow^* (\!|e', M'|\!)_i$. Then $\lfloor M'(m^{\beta p})\rfloor_1 = \lfloor M'(m^{\beta p})\rfloor_2 = M'(m^{\beta p})$.*

Proof: By Theorem A.5 we have $m^{\beta p} \in \text{dom}(M')$. A value of type β_p is either of the form $\langle v_1 \mid v_2\rangle$ or v_3, where v_1, v_2, v_3 are of type β_p and do not contain brackets. If $M'(m^{\beta p})$ is of the form $\langle v_1 \mid v_2\rangle$, then by (T-BRACKET) $H \trianglelefteq p$, a contradiction. Therefore $M'(m^{\beta p})$ must be of the form v_3, and the result holds. ∎

Proof of Theorem 5.4: Let e be a λ_{declass} expression such that $pc, \emptyset[x \mapsto \mathbf{int}_H] \vdash e : \tau$ for some security policy pc and type τ, where H is the security policy $\ell_1 \overset{c_1}{\rightsquigarrow} \cdots \overset{c_{k-1}}{\rightsquigarrow} \ell_k \overset{c_k}{\rightsquigarrow} p_H$. Let $\ell \in \mathcal{L}$ be an arbitrary security level such that $\ell_i \not\sqsubseteq_{\mathcal{L}} \ell$ for all $i \in 1..k$. Let v_1 and v_2 be two integer values. Let M be a memory such that $(\!|e[v_1/x], M|\!)$ and $(\!|e[v_2/x], M|\!)$ are well-formed and well-typed. Consider the $\lambda^2_{\text{declass}}$ configuration $(\!|e[\langle v_1 \mid v_2\rangle/x], M|\!)_\bullet$. This configuration is well-formed and well-typed, as $H \trianglelefteq H$. Let E_1 and E_2 be $c_1 \ldots c_k$-free λ_{declass} evaluations of $(\!|e[v_1/x], M|\!)$ and $(\!|e[v_2/x], M|\!)$ respectively. By adequacy (Lemma A.4) we have a $c_1 \ldots c_k$-free $\lambda^2_{\text{declass}}$ evaluation $(\!|e[\langle v_1 \mid v_2\rangle/x], M|\!)_\bullet (\!|e_1, M_1|\!)_\bullet \ldots (\!|e_n, M_n|\!)_\bullet$ such that $(\!|\lfloor e_n\rfloor_i, \lfloor M_n\rfloor_i|\!)$ is the last configuration of E_i, for some $i \in \{1, 2\}$.

Now, let $j \in 1..n$, and let $m^{\beta p}$ be any memory location in dom$(M_j|\ell)$. From the definition of dom$(M_j|\ell)$ we have $p \leq \ell \overset{\mathbf{f}}{\rightsquigarrow} \top$, so $p = \ell' \overset{c'}{\rightsquigarrow} p'$, for some ℓ' such that $\ell' \sqsubseteq_{\mathcal{L}} \ell$. It cannot be the case that $\ell_i \sqsubseteq_{\mathcal{L}} \ell'$ for some $i \in 1..k$, since this would imply $\ell_i \sqsubseteq_{\mathcal{L}} \ell$, which is a contradiction. So $H \not\trianglelefteq p$, and by Lemma A.6 the value $M_j(m^{\beta p})$ does not contain a bracket expression.

Thus, for all $j \in 1..n$, we have $\lfloor M_j\rfloor_1 \approx_\ell \lfloor M_j\rfloor_2$, and so $[E_i]_{\approx_\ell}$ is a prefix up to stuttering of $[E_j]_{\approx_\ell}$, where $\{i, j\} = \{1, 2\}$ and $(\!|\lfloor e_n\rfloor_i, \lfloor M_n\rfloor_i|\!)$ is the last configuration of E_i. Thus e is noninterfering for x at level ℓ until conditions $c_1, \ldots, c_k$. ∎