Nonmalleable Information Flow Control

Ethan Cecchetti
Department of Computer Science
Cornell University
ethan@cs.cornell.edu

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

Owen Arden
Department of Computer Science
University of California, Santa Cruz*
owen@soe.ucsc.edu

ABSTRACT

Noninterference is a popular semantic security condition because it offers strong end-to-end guarantees, it is inherently compositional, and it can be enforced using a simple security type system. Unfortunately, it is too restrictive for real systems. Mechanisms for downgrading information are needed to capture real-world security requirements, but downgrading eliminates the strong compositional security guarantees of noninterference.

We introduce *nonmalleable information flow*, a new formal security condition that generalizes noninterference to permit controlled downgrading of both confidentiality and integrity. While previous work on robust declassification prevents adversaries from exploiting the downgrading of confidentiality, our key insight is *transparent endorsement*, a mechanism for downgrading integrity while defending against adversarial exploitation. Robust declassification appeared to break the duality of confidentiality and integrity by making confidentiality depend on integrity, but transparent endorsement makes integrity depend on confidentiality, restoring this duality. We show how to extend a security-typed programming language with transparent endorsement and prove that this static type system enforces nonmalleable information flow, a new security property that subsumes robust declassification and transparent endorsement. Finally, we describe an implementation of this type system in the context of Flame, a flow-limited authorization plugin for the Glasgow Haskell Compiler.

CCS CONCEPTS

• **Security and privacy → Information flow control**;

Keywords: Downgrading; Information security; Security types

1 INTRODUCTION

An ongoing foundational challenge for computer security is to discover rigorous—yet sufficiently flexible—ways to specify what it means for a computing system to be secure. Such security conditions should be *extensional*, meaning that they are based on the externally observable behavior of the system rather than on unobservable details of its implementation. To allow security enforcement mechanisms to scale to large systems, a security condition should also be *compositional*, so that secure subsystems remain secure when combined into a larger system.

Noninterference, along with many variants [19, 35], has been a popular security condition precisely because it is both extensional and compositional. Noninterference forbids all flows of information from "high" to "low", or more generally, flows of information that violate a lattice policy [14].

Unfortunately, noninterference is also known to be too restrictive for most real systems, which need fine-grained control over when and how information flows. Consequently, most implementations of information flow control introduce *downgrading* mechanisms to allow information to flow contrary to the lattice policy. Downgrading confidentiality is called *declassification*, and downgrading integrity—that is, treating information as more trustworthy than information that has influenced it—is known as *endorsement* [47].

Once downgrading is permitted, noninterference is lost. The natural question is whether downgrading can nevertheless be constrained to guarantee that systems still satisfy some meaningful, extensional, and compositional security conditions. This paper shows how to constrain the use of both declassification and endorsement in a way that ensures such a security condition holds.

Starting with the work of Biba [7], integrity has often been viewed as dual to confidentiality. Over time, that simple duality has eroded. In particular, work on *robust declassification* [6, 11, 27, 46, 47] has shown that in the presence of declassification, confidentiality depends on integrity. It is dangerous to give the adversary the ability to influence declassification, either by affecting the data that is declassified or by affecting the decision to perform declassification. By preventing such influence, robust declassification stops the adversary from *laundering* confidential data through existing declassification operations. Operationally, languages prevent laundering by restricting declassification to high integrity program points. Robust declassification can be enforced using a modular type system and is therefore compositional.

This paper introduces a new security condition, *transparent endorsement*, which is dual to robust declassification: it controls endorsement by using *confidentiality* to limit the possible relaxations of *integrity*. Transparent endorsement prevents an agent from endorsing information that the provider of the information could not have seen. Such endorsement is dangerous because it permits the provider to affect flows from the endorser's own secret information into trusted information. This restriction on endorsement enforces an often-implicit justification for endorsing untrusted inputs in high-integrity, confidential computation (e.g., a password checker): low-integrity inputs chosen by an attacker should be chosen without knowledge of secret information.

A similar connection between the confidentiality and integrity of information arises in cryptographic settings. A *malleable* encryption scheme is one where a ciphertext encrypting one value can

*Work done while author was at Harvard University.

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134054

be transformed into a ciphertext encrypting a related value. While sometimes malleability is intentional (e.g., *homomorphic* encryption), an attacker's ability to generate ciphertexts makes malleable encryption insufficient to authenticate messages or validate integrity. Nonmalleable encryption schemes [15] prevent such attacks. In this paper, we combine robust declassification and transparent endorsement into a new security condition, *nonmalleable information flow*, which prevents analogous attacks in an information flow control setting.

The contributions of this paper are as follows:

- We give example programs showing the need for a security condition that controls endorsement of secret information.
- We generalize *robust declassification* to programs including complex data structures with heterogeneously labeled data.
- We identify *transparent endorsement* and *nonmalleable information flow*, new extensional security conditions for programs including declassification and endorsement.
- We present a core language, NMIFC, which provably enforces robust declassification, transparent endorsement, and nonmalleable information flow.
- We present the first formulation of robust declassification as a *4-safety hyperproperty*, and define two new 4-safety hyperproperties for transparent endorsement and nonmalleable information flow, the first time information security conditions have been characterized as k-safety hyperproperties with $k > 2$.
- We describe our implementation of NMIFC using Flame, a flow-limited authorization library for Haskell and adapt an example of the Servant web application framework, accessible online at http://memo.flow.limited.

We organize the paper as follows. Section 2 provides examples of vulnerabilities in prior work. Section 3 reviews relevant background. Section 4 introduces our approach for controlling dangerous endorsements, and Section 5 presents a syntax, semantics, and type system for NMIFC. Section 6 formalizes our security conditions and Section 7 restates them as hyperproperties. Section 8 discusses our Haskell implementation, Section 9 compares our approach to related work, and Section 10 concludes.

2 MOTIVATION

To motivate the need for an additional security condition and give some intuition about transparent endorsement, we give three short examples. Each example shows code that type-checks under existing information-flow type systems even though it contains insecure information flows, which we are able to characterize in a new way.

These examples use the notation of the flow-limited authorization model (FLAM) [4], which offers an expressive way to state both information flow restrictions and authorization policies. However, the problems observed in these examples are not specific to FLAM; they arise in all previous information-flow models that support downgrading (e.g., [8, 16, 22, 26, 33, 43, 48]). The approach in this paper can be applied straightforwardly to the decentralized label model (DLM) [26], and with more effort, to DIFC models that are less similar to FLAM. While some previous models lack a notion of integrity, from our perspective they are even worse off, because they effectively allow *unrestricted* endorsement.

```
1 String_T password;
2
3 boolean_{T←} check_password(String_{T→} guess) {
4     boolean_T endorsed_guess = endorse(guess, T);
5     boolean_T result = (endorsed_guess == password);
6     return declassify(result, T^←);
7 }
```

Figure 1: A password checker with malleable information flow

In FLAM, principals and information flow labels occupy the same space. Given a principal (or label) p, the notation $p^{\rightarrow}$ denotes the confidentiality projection of p, whereas the notation $p^{\leftarrow}$ denotes its integrity projection. Intuitively, $p^{\rightarrow}$ represents the authority to decide where p's secrets may flow *to*, whereas $p^{\leftarrow}$ represents the authority to decide where information trusted by p may flow *from*. Robust declassification ensures that the label $p^{\rightarrow}$ can be removed via declassification only in code that is trusted by p; that is, with integrity $p^{\leftarrow}$.

Information flow policies provide a means to specify security requirements for a program, but not an enforcement mechanism. For example, confidentiality policies might be implemented using encryption and integrity policies using digital signatures. Alternatively, hardware security mechanisms such as memory protection might be used to prevent untrusted processes from reading confidential data. The following examples illustrate issues that would arise in many information flow control systems, regardless of the enforcement mechanism.

2.1 Fooling a password checker

Password checkers are frequently used as an example of necessary and justifiable downgrading. However, incorrect downgrading can allow an attacker who does not know the password to authenticate anyway. Suppose there are two principals, a fully trusted principal T and an untrusted principal U. The following information flows are then secure: $U^{\rightarrow} \sqsubseteq T^{\rightarrow}$ and $T^{\leftarrow} \sqsubseteq U^{\leftarrow}$. Figure 1 shows in pseudo-code how we might erroneously implement a password checker in a security-typed language like Jif [25]. Because this pseudo-code would satisfy the type system, it might appear to be secure.

The argument `guess` has no integrity because it is supplied by an untrusted, possibly adversarial source. It is necessary to declassify the result of the function (at line 6) because the result indeed leaks a little information about the password. Robust declassification, as enforced in Jif, demands that the untrusted `guess` be endorsed before it can influence information released by declassification.

Unfortunately, the `check_password` policy does not prevent faulty or malicious (but well-typed) code from supplying `password` directly as the argument, thereby allowing an attacker with no knowledge of the correct password to "authenticate." Because `guess` is labeled as secret ($T^{\rightarrow}$), a flow of information from `password` to `guess` looks secure to the type system, so this severe vulnerability could remain undetected. To fix this we would need to make `guess` less secret, but no prior work has defined rules that would require this change. The true insecurity, however, lies on line 4, which erroneously treats sensitive information as if the attacker had constructed it. We can prevent this insecurity by outlawing such endorsements.

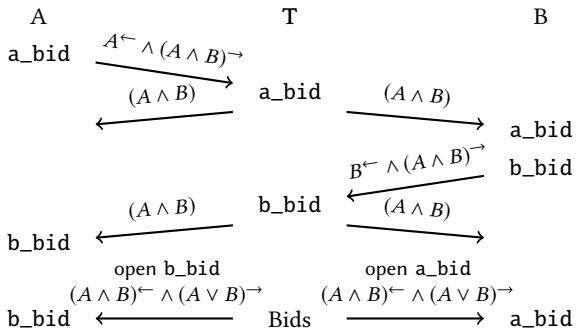

Figure 2: Cheating in a sealed-bid auction. Without knowing Alice's bid, Bob can always win by setting `b_bid := a_bid + 1`

2.2 Cheating in a sealed-bid auction

Imagine that two principals A and B (Alice and Bob) are engaging in a two-party sealed-bid auction administered by an auctioneer T whom they both trust. Such an auction might be implemented using cryptographic commitments and may even simulate T without need of an actual third party. However, we abstractly specify the information security requirements that such a scheme would aim to satisfy. Consider the following sketch of an auction protocol, illustrated in Figure 2:

(1) A sends her bid `a_bid` to T with label $A^{\leftarrow} \wedge (A \wedge B)^{\rightarrow}$. This label means `a_bid` is trusted only by those who trust A and can be viewed only if *both* A and B agree to release it.
(2) T accepts `a_bid` from A and uses his authority to endorse the bid to label $(A \wedge B)^{\leftarrow} \wedge (A \wedge B)^{\rightarrow}$ (identically, $A \wedge B$). The endorsement prevents any further unilateral modification to the bid by A. T then broadcasts this endorsed `a_bid` to A and B. This broadcast corresponds to an assumption that network messages can be seen by all parties.
(3) B constructs `b_bid` with label $B^{\leftarrow} \wedge (A \wedge B)^{\rightarrow}$ and sends it to T.
(4) T endorses `b_bid` to $A \wedge B$ and broadcasts the result.
(5) T now uses its authority to declassify both bids and send them to all parties. Since both bids have high integrity, this declassification is legal according to existing typing rules introduced to enforce (qualified) robust declassification [4, 11, 27].

Unfortunately, this protocol is subject to attacks analogous to *mauling* in malleable cryptographic schemes [15]: B can always win the auction with the minimal winning bid. In Step 3 nothing prevents B from constructing `b_bid` by adding 1 to `a_bid`, yielding a new bid with label $B^{\leftarrow} \wedge (A \wedge B)^{\rightarrow}$ (to modify the value, B must lower the value's integrity as A did not authorize the modification).

Again an insecurity stems from erroneously endorsing overly secret information. In step 4, T should not endorse `b_bid` since it could be based on confidential information inaccessible to B—in particular, `a_bid`. The problem can be fixed by giving A's bid the label $A^{\rightarrow} \wedge A^{\leftarrow}$ (identically, just A), but existing information flow systems impose no such requirement.

2.3 Laundering secrets

Wittbold and Johnson [44] present an interesting but insecure program:

```
1 while (true) do {
2     x = 0 [] x = 1; // generate secret probabilistically
3     output x to H;
4     input y from H; // implicit endorsement
5     output x ⊕ (y mod 2) to L
6 }
```

In this code, there are two external agents, H and L. Agent H is intended to have access to secret information, whereas L is not. The code generates a secret by assigning to the variable `x` a nondeterministic, secret value that is either 0 or 1. The choice of `x` is assumed not to be affected by the adversary. Its value is used as a one-time pad to conceal the secret low bit of variable `y`.

Wittbold and Johnson observe that this code permits an adversary to launder one bit of another secret variable `z` by sending `z⊕x` as the value read into `y`. The low bit of `z` is then the output to L.

Let us consider this classic example from the viewpoint of a modern information-flow type system that enforces robust declassification. In order for this code to type-check, it must declassify the value `x⊕(y mod 2)`. Since the attack depends on `y` being affected by adversarial input from H, secret input from H must be low-integrity (that is, its label must be $H^{\rightarrow}$). But if it is low-integrity, this input (or the variable `y`) must be endorsed to allow the declassification it influences. As in the previous two examples, the endorsement of high-confidentiality information enables exploits.

3 BACKGROUND

We explore nonmalleable information flow in the context of a simplified version of FLAM [4], so we first present some background. FLAM provides a unified model for reasoning about both information flow and authorization. Unlike in previous models, principals and information flow labels in FLAM are drawn from the same set $\mathcal{L}$. The interpretation of a label as a principal is the least powerful principal trusted to enforce that label. The interpretation of a principal as a label is the strongest information security policy that principal is trusted to enforce. We refer to elements of $\mathcal{L}$ as principals or labels depending on whether we are talking about authorization or information flow.

Labels (and principals) have both confidentiality and integrity aspects. A label (or principal) ℓ can be projected to capture just its confidentiality ($\ell^{\rightarrow}$) and integrity ($\ell^{\leftarrow}$) aspects.

The information flow ordering $\sqsubseteq$ on labels (and principals) describes information flows that are secure, in the direction of increasing confidentiality and decreasing integrity. The orthogonal trust ordering $\succcurlyeq$ on principals (and labels) corresponds to increasing trustedness and privilege: toward increasing confidentiality and *increasing* integrity. We read $\ell \sqsubseteq \ell'$ as "ℓ flows to ℓ'", meaning ℓ' specifies a policy at least as restrictive as ℓ does. We read $p \succcurlyeq q$ as "p acts for q", meaning that q delegates to p.

The information flow and the trust orderings each define a lattice over $\mathcal{L}$, and these lattices lie intuitively at right angles to one another. The least trusted and least powerful principal is $\bot$, (that is, $p \succcurlyeq \bot$ for all principals p), and the most trusted and powerful principal is $\top$ (where $\top \succcurlyeq p$ for all p). We also assume there is a set of *atomic principals* like `alice` and `bob` that define their own delegations.

Since the trust ordering defines a lattice, it has meet and join operations. Principal $p \wedge q$ is the least powerful principal that can act for both p and q; conversely, $p \vee q$ can act for all principals that both p and q can act for. The least element in the information flow ordering is $\top^{\leftarrow}$, representing maximal integrity and minimal confidentiality, whereas the greatest element is $\top^{\rightarrow}$, representing minimal integrity and maximal confidentiality. The join and meet operators in the information flow lattice are the usual $\sqcup$ and $\sqcap$, respectively.

Any principal (label) can be expressed in a normal form $p^{\rightarrow} \wedge q^{\leftarrow}$ where p and q are CNF formulas over atomic principals [4]. This normal form allows us to decompose decisions about lattice ordering (in either lattice) into separate questions regarding the integrity component (p) and the confidentiality component (q). Lattice operations can be similarly decomposed.

FLAM also introduces the concept of the *voice* of a label (principal) ℓ, written $\nabla(\ell)$. Formally, for a normal-form label $\ell = p^{\rightarrow} \wedge q^{\leftarrow}$, we define voice as follows: $\nabla(p^{\rightarrow} \wedge q^{\leftarrow}) \triangleq p^{\leftarrow}$.[1] A label's voice represents the minimum integrity needed to securely declassify data constrained by that label, a restriction designed to enforce robust declassification.

The Flow-Limited Authorization Calculus (FLAC) [5] previously embedded a simplified version of the FLAM proof system into a core language for enforcing secure authorization and information flow. FLAC is an extension of the Dependency Core Calculus (DCC) [1, 3] whose types contain FLAM labels. A computation is additionally associated with a program-counter label *pc* which tracks the influences on the control flow and values that are not explicitly labeled.

In this paper we take a similar approach: NMIFC enforces security policies by performing computation in a monadic context. As in FLAC, NMIFC includes a *pc* label. For an ordinary value v, the monadic term $(\eta_\ell\ v)$ signifies that value with the information flow label ℓ. If value v has type τ, the term $(\eta_\ell\ v)$ has type ℓ `says` τ, capturing the confidentiality and integrity of the information.

Unlike FLAC, NMIFC has no special support for dynamic delegation of authority. Atomic principals define $\mathcal{L}$ by statically delegating their authority to arbitrary conjunctions and disjunctions of other principals, and we include traditional declassification and endorsement operations, `decl` and `endorse`. We leave to future work the integration of nonmalleable information flow with secure dynamic delegation.

4 ENFORCING NONMALLEABILITY

Multiple prior security-typed languages—both functional [5] and imperative [6, 11, 27]—aim to allow some form of secure downgrading. These languages place no restriction whatsoever on the confidentiality of endorsed data or the context in which an endorsement occurs. Because of this permissiveness, all three insecure examples from Section 2 type-check in these languages.

4.1 Robust declassification

Robust declassification prevents adversaries from using declassifications in the program to release information that was not intended to be released. The adversary is assumed to be able to observe some state of the system, whose confidentiality label is sufficiently low, and to modify some state of the system, whose integrity label is sufficiently low. Semantically, robust declassification says that if the attacker is unable to learn a secret with one attack, no other attack will cause it to be revealed [27, 46]. The attacker has no control over information release because all attacks are equally good. When applied to a decentralized system, robust declassification means that for any principal p, other principals that p does not trust cannot influence declassification of p's secrets [11].

To enforce robust declassification, prior security-typed languages place integrity constraints on declassification. The original work on FLAM enforces robust declassification using the voice operator ∇. However, when declassification is expressed as a programming-language operation, as is more typical, it is convenient to define a new operator on labels, one that maps in the other direction, from integrity to confidentiality. We define the *view* of a principal as the upper bound on the confidentiality a label or context can enforce to securely endorse that label:

Definition 4.1 (Principal view). Let $\ell = p^{\rightarrow} \wedge q^{\leftarrow}$ be a FLAM label (principal) expressed in normal form. The *view* of ℓ, written $\Delta(\ell)$, is defined as $\Delta(p^{\rightarrow} \wedge q^{\leftarrow}) \triangleq q^{\rightarrow}$.

When the confidentiality of a label ℓ lies above the view of its own integrity, a declassification of that label may give adversaries the opportunity to subvert the declassification to release information. Without enough integrity, an adversary might, for example, replace the information that is intended to be released via declassification with some other secret.

Figure 3 illustrates this idea graphically. It depicts the lattice of FLAM labels, which is a product lattice with two axes, confidentiality and integrity. A given label ℓ is a point in this diagram, whereas the set of labels sharing the same confidentiality $\ell^{\rightarrow}$ or integrity $\ell^{\leftarrow}$ correspond to lines on the diagram. Given the integrity $\ell^{\leftarrow}$ of the label ℓ, the view of that integrity, $\Delta(\ell^{\leftarrow})$, defines a region of information (shaded) that is too confidential to be declassified.

The view operator directly corresponds to the writers-to-readers operator that Chong and Myers [11] use to enforce robust declassification in the DLM. We generalize the same idea here to the more expressive labels of FLAM.

4.2 Transparent endorsement

The key insight of this work is that endorsement should be restricted in a manner dual to robust declassification; declassification (reducing confidentiality) requires a minimum integrity, so endorsement (raising integrity) should require a *maximum* confidentiality. Intuitively, if a principal could have written data it cannot read, which we call an "opaque write," it is unsafe to endorse that data. An endorsement is *transparent* if it endorses only information its authors could read.

The voice operator suffices to express this new restriction conveniently, as depicted in Figure 4. In the figure, we consider endorsing information with confidentiality $\ell^{\rightarrow}$. This confidentiality is mapped to a corresponding integrity level $\nabla(\ell^{\rightarrow})$, defining a minimal integrity level that ℓ must have in order to be endorsed. If ℓ lies below this boundary, its endorsement is considered transparent; if it lies above the boundary, endorsement is *opaque* and hence insecure. The duality with robust declassification is clear.

[1] FLAM defines $\nabla(p^{\rightarrow} \wedge q^{\leftarrow}) = p^{\leftarrow} \wedge q^{\leftarrow}$, but our simplified definition is sufficient for NMIFC. For clarity, the operator ∇ is always applied to a projected principal.

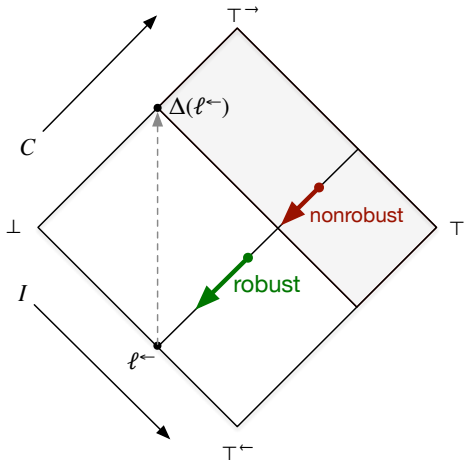

Figure 3: Robust declassification says information at level ℓ can be declassified only if it has enough integrity. The gray shaded region represents information that $\Delta(\ell^{\leftarrow})$ cannot read, so it is unsafe to declassify with ℓ's integrity.

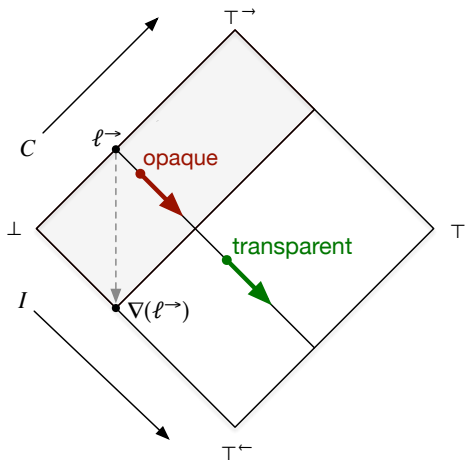

Figure 4: Transparent endorsement in NMIFC. The gray shaded region represents information that $\nabla(\ell^{\rightarrow})$ does not trust and may have been created by an opaque write. It is thus unsafe to endorse with ℓ's confidentiality.

5 A CORE LANGUAGE: NMIFC

We now describe the NonMalleable Information Flow Calculus (NMIFC), a new core language, modeled on DCC and FLAC, that allows downgrading, but in a more constrained manner than FLAC so as to provide stronger semantic guarantees. NMIFC incorporates the program-counter label *pc* of FLAC, but eschews the more powerful `assume` mechanism of FLAC in favor of more traditional declassify and endorse operations.

The full NMIFC is a small extension of Polymorphic DCC [1]. In Figure 5 we present the core syntax, leaving other features such as sums, pairs, and polymorphism to Appendix A. Unlike DCC, NMIFC supports downgrading and models it as an effect. It is necessary

$$
\begin{array}{lll}
n \in \mathcal{N} & & \text{(atomic principals)} \\
x \in \mathcal{V} & & \text{(variable names)} \\
\pi \in \{\rightarrow, \leftarrow\} & & \text{(security aspects)} \\
\\
p, \ell, pc & ::= & n \mid \top \mid \bot \mid p^{\pi} \mid p \wedge p \mid p \vee p \mid p \sqcup p \mid p \sqcap p \\
\tau & ::= & \texttt{unit} \mid \tau \xrightarrow{pc} \tau \mid \ell \ \texttt{says}\ \tau \\
v & ::= & () \mid \lambda(x\!:\!\tau)[pc].\, e \mid (\overline{\eta}_{\ell}\ v) \\
e & ::= & x \mid v \mid e\ e \mid (\eta_{\ell}\ e) \mid \texttt{bind}\ x = e\ \texttt{in}\ e \\
& \mid & \texttt{decl}\ e\ \texttt{to}\ \ell \mid \texttt{endorse}\ e\ \texttt{to}\ \ell
\end{array}
$$

Figure 5: Core NMIFC syntax.

$$\boxed{e \longrightarrow e'}$$

$$\text{[E-App]} \qquad (\lambda(x\!:\!\tau)[pc].\, e)\ v \longrightarrow e[x \mapsto v]$$

$$\text{[E-BindM]} \qquad \texttt{bind}\ x = (\overline{\eta}_{\ell}\ v)\ \texttt{in}\ e \longrightarrow e[x \mapsto v]$$

$$
\begin{array}{lll}
\text{(event)} & c ::= \bullet \mid (\overline{\eta}_{\ell}\ v) \mid (\downarrow^{\pi}_{\ell'}, \overline{\eta}_{\ell}\ v) \\
\text{(trace)} & t ::= \varepsilon \mid c \mid t; t
\end{array}
$$

$$\boxed{\langle e,\ t\rangle \longrightarrow\!\!\!\!\rightarrow \langle e',\ t'\rangle}$$

$$\text{[E-Step]} \qquad \frac{e \longrightarrow e'}{\langle e,\ t\rangle \longrightarrow\!\!\!\!\rightarrow \langle e',\ t;\bullet\rangle}$$

$$\text{[E-UnitM]} \qquad \langle (\eta_{\ell}\ v),\ t\rangle \longrightarrow\!\!\!\!\rightarrow \big\langle (\overline{\eta}_{\ell}\ v),\ t; (\overline{\eta}_{\ell}\ v)\big\rangle$$

$$\text{[E-Decl]} \qquad \big\langle \texttt{decl}\ (\overline{\eta}_{\ell'}\ v)\ \texttt{to}\ \ell,\ t\big\rangle \longrightarrow\!\!\!\!\rightarrow \big\langle (\overline{\eta}_{\ell}\ v),\ t; (\downarrow^{\rightarrow}_{\ell'}, \overline{\eta}_{\ell}\ v)\big\rangle$$

$$\text{[E-Endorse]} \qquad \big\langle \texttt{endorse}\ (\overline{\eta}_{\ell'}\ v)\ \texttt{to}\ \ell,\ t\big\rangle \longrightarrow\!\!\!\!\rightarrow \big\langle (\overline{\eta}_{\ell}\ v),\ t; (\downarrow^{\leftarrow}_{\ell'}, \overline{\eta}_{\ell}\ v)\big\rangle$$

$$\text{[E-Eval]} \qquad \frac{\langle e,\ t\rangle \longrightarrow\!\!\!\!\rightarrow \langle e',\ t'\rangle}{\langle E[e],\ t\rangle \longrightarrow\!\!\!\!\rightarrow \langle E[e'],\ t'\rangle}$$

Evaluation context

$$
\begin{array}{lll}
E & ::= & [\cdot] \mid E\ e \mid v\ E \mid (\eta_{\ell}\ E) \mid \texttt{bind}\ x = E\ \texttt{in}\ e \\
& \mid & \texttt{decl}\ E\ \texttt{to}\ \ell \mid \texttt{endorse}\ E\ \texttt{to}\ \ell
\end{array}
$$

Figure 6: Core NMIFC operational semantics.

to track what information influences control flow so that these downgrading effects may be appropriately constrained. Therefore, like FLAC, NMIFC adds *pc* labels to lambda terms and types.

Similarly to DCC, protected values have type ℓ `says` τ where ℓ is the confidentiality and integrity of a value of type τ. All computation on these values occurs in the `says` monad; protected values must be bound using the `bind` term before performing operations on them (e.g., applying them as functions). Results of such computations are protected with the monadic unit operator $(\eta_{\ell}\ e)$, which protects the result of e with label ℓ.

5.1 NMIFC operational semantics

The core semantics of NMIFC are mostly standard, but to obtain our theoretical results we need additional information about evaluation. This information is necessary because we want to identify, for instance, whether information is ever available to an attacker during evaluation, even if it is discarded and does not influence the final

$\boxed{\vdash \ell \sqsubseteq \tau}$

[P-Unit] $\vdash \ell \sqsubseteq \texttt{unit}$ [P-Lbl] $\dfrac{\ell' \sqsubseteq \ell}{\vdash \ell' \sqsubseteq \ell \texttt{ says } \tau}$

Figure 7: Type protection levels.

result. This approach gives an attacker more power; an attacker can see information at its level even if it is not output by the program.

The NMIFC semantics, presented in Figure 6, maintain a trace t of events. An event is emitted into the trace whenever a new protected value is created and whenever a declassification or endorsement occurs. These events track the observations or influence an attacker may have during a run of an NMIFC program. Formally, a trace can be an empty trace ε, a single event e, or the concatenation of two traces with the associative operator ";" with identity ε.

When a source-level unit term $(\eta_\ell\ v)$ is evaluated (rule E-UnitM), an event $(\overline{\eta}_\ell\ v)$ is added to the trace indicating that the value v became protected at ℓ. When a protected value is declassified, a declassification event $(\downarrow^{\rightarrow}_{\ell'}, \overline{\eta}_\ell\ v)$ is emitted, indicating that v was declassified from ℓ' to ℓ. Likewise, an endorsement event $(\downarrow^{\leftarrow}_{\ell'}, \overline{\eta}_\ell\ v)$ is emitted for an endorsement. Other evaluation steps (rule E-Step) emit $\bullet$, for "no event." Rule E-Eval steps under the evaluation contexts [45] defined at the bottom of Figure 6.

Rather than being literal side effects of the program, these events track how observable information is as it is accessed, processed, and protected by the program. Because our semantics emits an event whenever information is protected (by evaluating an η term) or downgraded (by a `decl` or `endorse` term), our traces capture all information processed by a program, indexed by the policy protecting that information.

By analogy, these events are similar to the typed and labeled mutable reference cells of languages like FlowCaml [31] and DynSec [49]. An event $(\overline{\eta}_\ell\ v)$ is analogous to allocating a reference cell protected at ℓ, and $(\downarrow^{\pi}_{\ell'}, \overline{\eta}_\ell\ v)$ is analogous to copying the contents of a cell at ℓ' to a new cell at ℓ.

It is important for the semantics to keep track of these events so that our security conditions hold for programs containing data structures and higher-order functions. Previous language-based definitions of robust declassification have only applied to simple `while`-languages [6, 11, 27] or to primitive types [5].

5.2 NMIFC type system

The NMIFC protection relation, presented in Figure 7, defines how types relate to information flow policies. A type τ protects the confidentiality and integrity of ℓ if $\vdash \ell \sqsubseteq \tau$. Unlike in DCC and FLAC, a label is protected by a type only if it flows to the outermost `says` principal. In FLAC and DCC, the types ℓ' `says` ℓ `says` τ and ℓ `says` ℓ' `says` τ protect the same set of principals; in other words, `says` is commutative. By distinguishing between these types, NMIFC does not provide the same commutativity.

The commutativity of `says` is a design decision, offering a more permissive programming model at the cost of less precise tracking of dependencies. NMIFC takes advantage of this extra precision in the UnitM typing rule so the label on every η term protects the information it contains, even if nested within other η terms.

$\boxed{\Gamma; pc \vdash e : \tau}$

[Var] $\Gamma, x:\tau, \Gamma'; pc \vdash x : \tau$ [Unit] $\Gamma; pc \vdash () : \texttt{unit}$

[Lam] $\dfrac{\Gamma, x:\tau_1; pc' \vdash e : \tau_2}{\Gamma; pc \vdash \lambda(x:\tau_1)[pc'].\ e : \tau_1 \xrightarrow{pc'} \tau_2}$ [App] $\dfrac{\Gamma; pc \vdash e_1 : \tau' \xrightarrow{pc'} \tau \quad \Gamma; pc \vdash e_2 : \tau' \quad pc \sqsubseteq pc'}{\Gamma; pc \vdash e_1\ e_2 : \tau}$

[UnitM] $\dfrac{\Gamma; pc \vdash e : \tau \quad pc \sqsubseteq \ell}{\Gamma; pc \vdash (\eta_\ell\ e) : \ell \texttt{ says } \tau}$ [VUnitM] $\dfrac{\Gamma; pc \vdash v : \tau}{\Gamma; pc \vdash (\overline{\eta}_\ell\ v) : \ell \texttt{ says } \tau}$

[BindM] $\dfrac{\Gamma; pc \vdash e : \ell \texttt{ says } \tau' \quad \vdash \ell \sqsubseteq \tau \quad \Gamma, x:\tau'; pc \sqcup \ell \vdash e' : \tau}{\Gamma; pc \vdash \texttt{bind } x = e \texttt{ in } e' : \tau}$

[Decl] $\dfrac{\Gamma; pc \vdash e : \ell' \texttt{ says } \tau \quad \ell'^{\leftarrow} = \ell^{\leftarrow} \quad pc \sqsubseteq \ell \quad \ell'^{\rightarrow} \sqsubseteq \ell^{\rightarrow} \sqcup \Delta((\ell' \sqcup pc)^{\leftarrow})}{\Gamma; pc \vdash \texttt{decl } e \texttt{ to } \ell : \ell \texttt{ says } \tau}$

[Endorse] $\dfrac{\Gamma; pc \vdash e : \ell' \texttt{ says } \tau \quad \ell'^{\rightarrow} = \ell^{\rightarrow} \quad pc \sqsubseteq \ell \quad \ell'^{\leftarrow} \sqsubseteq \ell^{\leftarrow} \sqcup \nabla((\ell' \sqcup pc)^{\rightarrow})}{\Gamma; pc \vdash \texttt{endorse } e \texttt{ to } \ell : \ell \texttt{ says } \tau}$

Figure 8: Typing rules for core NMIFC.

Abadi [2] similarly modifies DCC's protection relation to distinguish the protection level of terms with nested `says` types.

The core type system presented in Figure 8 enforces nonmalleable information flow for NMIFC programs. Most of the typing rules are standard, and differ only superficially from DCC and FLAC. Like in FLAC, NMIFC typing judgments include a program counter label, pc, that represents an upper bound on the confidentiality and integrity of bound information that any computation may depend upon. For instance, rule BindM requires the type of the body of a `bind` term to protect the unlabeled information of type τ' with at least ℓ, and to type-check under a raised program counter label $pc \sqcup \ell$. Rule Lam ensures that function bodies type-check with respect to the function's pc annotation, and rule App ensures functions are only applied in contexts that flow to these annotations.

The NMIFC rule for UnitM differs from FLAC and DCC in requiring the premise $pc \sqsubseteq \ell$ for well-typed η terms. This premise ensures a more precise relationship between the pc and η terms. Intuitively this restriction makes sense. The pc is a bound on all unlabeled information in the context. Since an expression e protected with $(\eta_\ell\ e)$ may depend on any of this information, it makes sense to require that pc flow to ℓ.[2]

By itself, this restrictive premise would prevent public data from flowing through secret contexts and trusted data from flowing through untrusted contexts. To allow such flows, we distinguish source-level $(\eta_\ell\ e)$ terms from run-time values $(\overline{\eta}_\ell\ v)$, which have been fully evaluated. These terms are only created by the operational semantics during evaluation and no longer depend on the context in which they appear; they are closed terms. Thus it is appropriate to omit the new premise in VUnitM. This approach allows us to require more precise flow tracking for the explicit dependencies of protected expressions without restricting where these values flow once they are fully evaluated.

[2] The premise is not required in FLAC because protection is commutative. For example, in a FLAC term such as `bind` $x = v$ `in` $(\eta_{\ell'}\ (\eta_\ell\ x))$, x may be protected by ℓ or ℓ'.

$$\texttt{checkpwd} = \lambda(g : U^{\leftarrow}\ \texttt{says String},\ p : T\ \texttt{says String})[T^{\leftarrow}].$$
$$\texttt{bind}\ \mathit{guess} = (\texttt{endorse}\ g\ \texttt{to}\ T^{\leftarrow})\ \texttt{in}$$
$$\texttt{decl}\ (\texttt{bind}\ \mathit{pwd} = p\ \texttt{in}\ (\eta_T\ \mathit{pwd} == \mathit{guess}))\ \texttt{to}\ T^{\leftarrow}$$

Figure 9: A secure version of a password checker.

Rule Decl ensures a declassification from label ℓ' to ℓ is robust. We first require $\ell'^{\leftarrow} = \ell^{\leftarrow}$ to ensure that this does not perform endorsement. A more permissive premise $\ell'^{\leftarrow} \sqsubseteq \ell^{\leftarrow}$ is admissible, but requiring equality simplifies our proofs and does not reduce expressiveness since the declassification can be followed by a subsequent relabeling. The premise $pc \sqsubseteq \ell$ requires that declassifications occur in high-integrity contexts, and prevents declassification events from creating implicit flows. The premise $\ell'^{\rightarrow} \sqsubseteq \ell^{\rightarrow} \sqcup \Delta((\ell' \sqcup pc)^{\leftarrow})$ ensures that the confidentiality of the information declassified does not exceed the view of the integrity of the principals that may have influenced it. These influences can be either explicit ($\ell'^{\leftarrow}$) or implicit ($pc^{\leftarrow}$), so we compare against the join of the two.[3] This last premise effectively combines the two conditions identified by Chong and Myers [11] for enforcing robust declassification in an imperative `while`-language.

Rule Endorse enforces transparent endorsement. All but the last premise are straightforward: the expression does not declassify and $pc \sqsubseteq \ell$ requires a high-integrity context to endorse and prevents implicit flows. Interestingly, the last premise is dual to that in Decl. An endorsement cannot raise integrity above the voice of the confidentiality of the data being endorsed ($\ell'^{\rightarrow}$) or the context performing the endorsement ($pc^{\rightarrow}$). For the same reasons as in Decl, we compare against their join.

5.3 Examples revisited

We now reexamine the examples presented in Section 2 to see that the NMIFC type system prevents the vulnerabilities seen above.

5.3.1 Password checker. We saw above that when the password checker labels `guess` at $T^{\rightarrow}$, well-typed code can improperly set `guess` to the actual password. We noted that the endorsement enabled an insecure flow of information. Looking at Endorse in NMIFC, we can attempt to type the equivalent expression: `endorse` *guess* `to` T. However, if *guess* has type $T^{\rightarrow}$ `says bool`, the `endorse` does not type-check; it fails to satisfy the final premise of Endorse:

$$\bot^{\leftarrow} = (T^{\rightarrow})^{\leftarrow} \not\sqsubseteq T^{\leftarrow} \sqcup \nabla(T^{\rightarrow}) = T^{\leftarrow}.$$

If we instead give `guess` the label $U^{\leftarrow}$, the endorsement type-checks, assuming a sufficiently trusted *pc*.

This is as it should be. With the label $U^{\leftarrow}$, the guesser must be able to read their own guess, guaranteeing that they cannot guess the correct password unless they in fact know the correct password. Figure 9 shows this secure version of the password checker.

5.3.2 Sealed-bid auction. In the insecure auction described in Section 2.2, we argued that an insecure flow was created when T endorsed `b_bid` from $B^{\leftarrow} \wedge (A \wedge B)^{\rightarrow}$ to $A \wedge B$. This endorsement requires a term of the form `endorse` v `to` $A \wedge B$ where v types to $B^{\leftarrow} \wedge (A \wedge B)^{\rightarrow}$ `says int`. Despite the trusted context, the last premise of Endorse again fails:

$$B^{\leftarrow} \not\sqsubseteq (A \wedge B)^{\leftarrow} \sqcup \nabla((A \wedge B)^{\rightarrow}) = (A \wedge B)^{\leftarrow}.$$

If we instead label `a_bid` : A `says int` and `b_bid` : B `says int`, then the corresponding `endorse` statements type-check, assuming that T is trusted: $T \sqsubseteq (A \wedge B)^{\leftarrow}$.

5.3.3 Laundering secrets. For the secret-laundering example in Section 2.3, we assume that neither H nor L is trusted, but the output from the program is. This forces an implicit endorsement of y, the input received from H. But the condition needed to endorse from $H^{\rightarrow} \wedge \bot^{\leftarrow}$ to $H^{\rightarrow} \wedge \top^{\leftarrow}$ is false:

$$\bot^{\leftarrow} \sqsubseteq \top^{\leftarrow} \sqcup \nabla(H^{\rightarrow}) = \nabla(H^{\rightarrow})$$

We have $\nabla(L^{\rightarrow}) \not\sqsubseteq \nabla(H^{\rightarrow})$ and all integrity flows to $\bot^{\leftarrow}$, so by transitivity the above condition cannot hold.

6 SECURITY CONDITIONS

The NMIFC typing rules enforce several strong security conditions: multiple forms of conditional noninterference, robust declassification, and our new transparent endorsement and nonmalleable information flow conditions. We define these conditions formally but relegate proof details to the technical report [10].

6.1 Attackers

Noninterference is frequently stated with respect to a specific but arbitrary label. Anything below that label in the lattice is "low" (public or trusted) and everything else is "high". We broaden this definition slightly and designate high information using a set of labels $\mathcal{H}$ that is upward closed. That is, if $\ell \in \mathcal{H}$ and $\ell \sqsubseteq \ell'$, then $\ell' \in \mathcal{H}$. We refer to such upward closed sets as *high sets*.

We say that a type τ is a *high type*, written "$\vdash \tau$ prot $\mathcal{H}$", if all of the information in a value of type τ is above some label in the high set $\mathcal{H}$. The following rule defines high types:

$$\text{[P-Set]} \qquad \frac{H \in \mathcal{H} \qquad \vdash H \sqsubseteq \tau}{\vdash \tau \text{ prot } \mathcal{H}}\ \mathcal{H} \text{ is upward closed}$$

This formulation of adversarial power is adequate to express noninterference, in which confidentiality and integrity do not interact. However, our more complex conditions relate confidentiality to integrity and therefore require a way to relate the attacker's power in the two domains.

Intuitively, an attacker is an arbitrary set of colluding atomic principals. Specifically, if $n_1, \ldots, n_k \in \mathcal{N}$ are those atomic principals, then the set $\mathcal{A} = \{\ell \in \mathcal{L} \mid n_1 \wedge \cdots \wedge n_k \succcurlyeq \ell\}$ represents this attacker's power. These principals may include principals mentioned in the program, and there may be delegations between attacker principals and program principals. While this definition of an attacker is intuitive, the results in this paper actually hold for a more general notion of attacker defined in Appendix B.

Attackers correspond to two high sets: an *untrusted* high set $\mathcal{U} = \{\ell \in \mathcal{L} \mid \ell^{\leftarrow} \in \mathcal{A}\}$ and a *secret* high set $\mathcal{S} = \{\ell \in \mathcal{L} \mid \ell^{\rightarrow} \notin \mathcal{A}\}$. We say that $\mathcal{A}$ *induces* $\mathcal{U}$ and $\mathcal{S}$.

[3] The first two premises—$\ell'^{\leftarrow} = \ell^{\leftarrow}$ and $pc \sqsubseteq \ell$—make this join redundant. It would, however, be necessary if we replaced the equality premise with the more permissive $\ell'^{\leftarrow} \sqsubseteq \ell^{\leftarrow}$ version, so we include it for clarity.

$\boxed{c \approx_{\mathcal{W}} c'}$ $\boxed{v \approx_{\mathcal{W}} v'}$

These equivalence relations are the smallest congruences over c and over v extended with $\bullet$, containing the equivalences defined by these rules:

[Eq-UnitM] $\dfrac{\ell \notin \mathcal{W}}{(\overline{\eta}_\ell\ v) \approx_{\mathcal{W}} \bullet}$ [Eq-Down] $\dfrac{\ell \notin \mathcal{W}}{(\downarrow^{\pi}_{\ell'},\ \overline{\eta}_\ell\ v) \approx_{\mathcal{W}} \bullet}$

$\boxed{t \approx^{\star}_{\mathcal{W}} t'}$

The equivalence relation $t \approx^{\star}_{\mathcal{W}} t'$ is the smallest congruence over t containing the equivalences defined by these rules:

[T-Lift] $\dfrac{c \approx_{\mathcal{W}} c'}{c \approx^{\star}_{\mathcal{W}} c'}$ [T-BulletR] $t; \bullet \approx^{\star}_{\mathcal{W}} t$ [T-BulletL] $\bullet; t \approx^{\star}_{\mathcal{W}} t$

Figure 10: Low equivalence and low trace equivalence.

6.2 Equivalences

All of our security conditions involve relations on traces. As is typically the case for information-flow security conditions, we define a notion of "low equivalence" on traces, which ignores effects with high labels. We proceed by defining low-equivalent expressions and then extending low-equivalence to traces.

For expression equivalence, we examine precisely the values which are visible to a low observer defined by a set of labels $\mathcal{W}$: $(\overline{\eta}_\ell\ v)$ and $(\downarrow^{\pi}_{\ell'},\ \overline{\eta}_\ell\ v)$ where $\ell \in \mathcal{W}$. We formalize this idea in Figure 10, using $\bullet$ to represent values that are not visible. Beyond ignoring values unable to affect the output, we use a standard structural congruence (i.e., syntactic equivalence). This strict notion of equivalence is not entirely necessary; observational equivalence or any refinement thereof would be sufficient if augmented with the $\bullet$-equivalences in Figure 10.

Figure 10 also extends the equivalence on emitted values to equivalence on entire traces of emitted values. Essentially, two traces are equivalent if there is a way to match up equivalent events in each trace, while ignoring high events equivalent to $\bullet$.

6.3 Noninterference and downgrading

An immediate consideration when formalizing information flow is how to express interactions between an adversary and the system. One possibility is to limit interaction to inputs and outputs of the program. This is a common approach for functional languages. We take a stronger approach in which security is expressed in terms of execution traces. Note that traces contain all information necessary to ensure the security of input and output values.

We begin with a statement of noninterference in the presence of downgrading. Theorem 6.1 states that, given two high inputs, a well-typed program produces two traces that are either low-equivalent or contain a downgrade event that distinguishes them. This implies that differences in traces distinguishable by an attacker are all attributable to downgrades of information derived from the high inputs. Furthermore, any program that performs no downgrades on secret or untrusted values (i.e., contain no `decl` or `endorse` terms on $\mathcal{H}$ data) must be noninterfering.

Theorem 6.1 (Noninterference modulo downgrading). *Let $\mathcal{H}$ be a high set and let $\mathcal{W} = \mathcal{L} \setminus \mathcal{H}$. Given an expression e such that $\Gamma, x{:}\tau_1; pc \vdash e : \tau_2$ where $\vdash \tau_1$ prot $\mathcal{H}$, for all v_1, v_2 with $\Gamma; pc \vdash v_i : \tau_1$, if*

$$\langle e[x \mapsto v_i],\ v_i\rangle \longrightarrow^{*} \left\langle v_i',\ t^i\right\rangle$$

then either there is some event $(\downarrow^{\pi}_{\ell'},\ \overline{\eta}_\ell\ w) \in t^i$ where $\ell' \in \mathcal{H}$ and $\ell \notin \mathcal{H}$, or $t^1 \approx^{\star}_{\mathcal{W}} t^2$.

The restrictions placed on downgrading operations mean that we can characterize the conditions under which no downgrading can occur. We add two further noninterference theorems that restrict downgrading in different ways. Theorem 6.2 states that if a program types without a public–trusted *pc* it must be noninterfering (with respect to that definition of "public–trusted").

Theorem 6.2 (Noninterference of high-*pc* programs). *Let $\mathcal{A}$ be an attacker inducing high sets $\mathcal{U}$ and $\mathcal{S}$. Let $\mathcal{H}$ be one of those high sets and $\mathcal{W} = \mathcal{L} \setminus \mathcal{H}$. Given some e such that $\Gamma, x{:}\tau_1; pc \vdash e : \tau_2$ where $\vdash \tau_1$ prot $\mathcal{H}$, for all v_1, v_2 with $\Gamma; pc \vdash v_i : \tau_1$, if $\langle e[x \mapsto v_i],\ v_i\rangle \longrightarrow^{*} \langle v_i',\ t^i\rangle$ and $pc \in \mathcal{U} \cup \mathcal{S}$, then $t^1 \approx^{\star}_{\mathcal{W}} t^2$.*

Rather than restrict the *pc*, Theorem 6.3 states that secret–untrusted information is *always* noninterfering. Previous work (e.g., [6, 27]) does not restrict endorsement of confidential information, allowing any label to be downgraded to public–trusted (given a public–trusted *pc*). In NMIFC, however, secret–untrusted data must remain secret and untrusted.

Theorem 6.3 (Noninterference of secret–untrusted data). *Let $\mathcal{A}$ be an attacker inducing high sets $\mathcal{U}$ and $\mathcal{S}$. Let $\mathcal{H} = \mathcal{U} \cap \mathcal{S}$ and $\mathcal{W} = \mathcal{L} \setminus \mathcal{H}$. Given some e such that $\Gamma, x{:}\tau_1; pc \vdash e : \tau_2$ where $\vdash \tau_1$ prot $\mathcal{H}$, for all v_1, v_2 with $\Gamma; pc \vdash v_i : \tau_1$, if $\langle e[x \mapsto v_i],\ v_i\rangle \longrightarrow^{*} \langle v_i',\ t^i\rangle$ then $t^1 \approx^{\star}_{\mathcal{W}} t^2$.*

6.4 Robust declassification and irrelevant inputs

We now move to security conditions for programs that do not satisfy noninterference. Recall that robust declassification informally means the attacker has no influence on what information is released by declassification. Traditionally, it is stated in terms of attacker-provided code that is inserted into low-integrity holes in programs which differ only in their secret inputs. In NMIFC, the same attacker power can be obtained by substituting exactly two input values into the program, one secret and one untrusted. This simplification is possible because NMIFC has first-class functions that can model the substitution of low-integrity code. Appendix C shows that this simpler two-input definition is equivalent to the traditional hole-based approach in the full version of NMIFC (Appendix A).

Prior work on `while`-based languages [11, 27] defines robust declassification in terms of four traces generated by the combination of two variations: a secret input and some attacker-supplied code. For terminating traces, these definitions require any pair of secrets to produce public-equivalent traces under all attacks or otherwise to produce distinguishable traces regardless of the attacks chosen. This implies that an attacker cannot control the disclosure of secrets.

We can attempt to capture this notion of robust declassifcation using the notation of NMIFC. For a program e with a secret input x and untrusted input y, we wish to say e robustly declassifies if, for all secret values v_1, v_2 and for all untrusted values w_1, w_2, where

$$\left\langle e[x \mapsto v_i][y \mapsto w_j],\ v_i; w_j\right\rangle \longrightarrow^{*} \left\langle v_{ij},\ t^{ij}\right\rangle,$$

then $t^{11} \approx^{\star}_{\mathcal{P}} t^{21} \iff t^{12} \approx^{\star}_{\mathcal{P}} t^{22}$.

$$
\begin{array}{l}
(\lambda(x : (P^{\rightarrow} \wedge U^{\leftarrow}\ \texttt{says}\ \tau) \times (P^{\rightarrow} \wedge U^{\leftarrow}\ \texttt{says}\ \tau))[P^{\rightarrow} \wedge T^{\leftarrow}]. \\
\quad \texttt{decl}\ (\texttt{bind}\ b = (\eta_{S^{\rightarrow} \wedge T^{\leftarrow}}\ \mathit{sec})\ \texttt{in} \\
\qquad \texttt{case}\ b\ \texttt{of}\ \texttt{inj}_1\ .(\eta_{S^{\rightarrow} \wedge T^{\leftarrow}}\ (\texttt{proj}_1\ x)) \\
\qquad\qquad \mid \texttt{inj}_2\ .(\eta_{S^{\rightarrow} \wedge T^{\leftarrow}}\ (\texttt{proj}_2\ x))) \\
\qquad \texttt{to}\ P^{\rightarrow} \wedge T^{\leftarrow})\ \langle atk_1, atk_2 \rangle
\end{array}
$$

Figure 11: A program that admits inept attacks. Here $P \sqsubseteq S$ and $T \sqsubseteq U$, but not vice versa, so *sec* is a secret boolean and $\langle atk_1, atk_2 \rangle$ form an untrusted pair of values. If $atk_1 \neq atk_2$, then the attacker will learn the value of *sec*. If $atk_1 = atk_2$, however, then the attacker learns nothing due to its own ineptness.

This condition is intuitive but, unfortunately, overly restrictive. It does not account for the possibility of an *inept attack*, in which an attacker causes a program to reveal less information than intended.

Inept attacks are harder to characterize than in previous work because, unlike the previously used `while`-languages, NMIFC supports data structures with heterogeneous labels. Using such data structures, we can build a program that implicitly declassifies data by using a secret to influence the selection of an attacker-provided value and then declassifying that selection. Figure 11 provides an example of such a program, which uses sums and products from the full NMIFC language.

While this program appears secure—the attacker has no control over what information is declassified or when a declassification occurs—it violates the above condition. One attack can contain the same value twice—causing any two secrets to produce indistinguishable traces—while the other can contain different values. Intuitively, no vulnerability in the program is thereby revealed; the program was *intended* to release information, but the attacker failed to infer it due to a poor choice of attack. Such inputs result in less information leakage entirely due to the attacker's ineptness, not an insecurity of the program. As a result, we consider inputs from inept attackers to be *irrelevant* to our security conditions.

Dually to inept attackers, we can define uninteresting secret inputs. For example, if a program endorses an attacker's selection of a secret value, an input where all secret options contain the same data is uninteresting, so we also consider it irrelevant.

Which inputs are irrelevant is specific to the program and to the choice of attacker. In Figure 11, if both execution paths used ($\texttt{proj}_1\ x$), there would be no way for an attacker to learn any information, so all attacks are equally relevant. Similarly, if $S^{\rightarrow}$ is already considered public, then there is no secret information in the first place, so again, all attacks are equally relevant.

For an input to be irrelevant, it must have no influence over the outermost layer of the data structure—the label that is explicitly downgraded. If the input could influence that outer layer in any way, the internal data could be an integral part of an insecure execution. Conversely, when the selection of nested values is independent of any untrusted/secret information (though the content of the values may not be), it is reasonable to assume that the inputs will be selected so that different choices yield different results. An input which does not is either an inept attack—an attacker gaining less information than it could have—or an uninteresting secret—a choice between secrets that are actually the same. In either case, the input is irrelevant.

To ensure that we only consider data structures with nested values that were selected independently of the values themselves, we leverage the noninterference theorems in Section 6.3. In particular, if the outermost label is trusted before a declassification (or public prior to an endorsement), then any influence from untrusted (secret) data must be the result of a prior explicit downgrade. Thus we can identify irrelevant inputs by finding inputs that result in traces that are public-trusted equivalent, but can be made both public (trusted) equivalent and non-equivalent at the point of declassification (endorsement).

To define this formally, we begin by partitioning the principal lattice into four quadrants using the definition of an attacker from Section 6.1. We consider only flows between quadrants and, as with noninterference, downgrades must result in public or trusted values. We additionally need to refer to individual elements and prefixes of traces. For a trace t, let t_n denote the nth element of t, and let $t_{..n}$ denote the prefix of t containing its first n elements.

Definition 6.4 (Irrelevant inputs). Consider attacker $\mathcal{A}$ inducing high sets $\mathcal{H}_{\leftarrow}$ and $\mathcal{H}_{\rightarrow}$. Let $\mathcal{W}_{\pi} = \mathcal{L} \setminus \mathcal{H}_{\pi}$ and $\mathcal{W} = \mathcal{W}_{\leftarrow} \cap \mathcal{W}_{\rightarrow}$. Given opposite projections π and π', a program e, and types τ_x and τ_y such that $\vdash \tau_x$ prot $\mathcal{H}_{\pi'}$ and $\vdash \tau_y$ prot $\mathcal{H}_{\pi}$, we say an input v_1 is an *irrelevant π'-input* with respect to $\mathcal{A}$ and e if $\Gamma; pc \vdash v_1 : \tau_x$ and there exist values v_2, w_1, and w_2 and four trace indices n_{ij} (for $i, j \in \{1, 2\}$) such that the following conditions hold:

(1) $\Gamma; pc \vdash v_2 : \tau_x$, $\Gamma; pc \vdash w_1 : \tau_y$, and $\Gamma; pc \vdash w_2 : \tau_y$
(2) $\langle e[x \mapsto v_i][y \mapsto w_j],\ v_i; w_j \rangle \longrightarrow\!\!\!\!\rightarrow^{*} \langle v_{ij},\ t^{ij} \rangle$
(3) $t^{ij}_{n_{ij}} \not\approx_{\mathcal{W}} \bullet$ for all $i, j \in \{1, 2\}$
(4) $t^{ij}_{..n_{ij}} \approx^{\star}_{\mathcal{W}} t^{kl}_{..n_{kl}}$ for all $i, j, k, l \in \{1, 2\}$
(5) $t^{11}_{..n_{11}} \approx^{\star}_{\mathcal{W}_{\pi}} t^{12}_{..n_{12}}$
(6) $t^{21}_{..n_{21}} \not\approx^{\star}_{\mathcal{W}_{\pi}} t^{22}_{..n_{22}}$

Otherwise we say v_1 is a *relevant π'-input* with respect to $\mathcal{A}$ and e, denoted $\mathrm{rel}^{\pi'}_{\mathcal{A}, e}(v_1)$. Note that the four indices n_{ij} identify corresponding prefixes of the four traces.

Note: this definition corrects a typographical error in the version published.

As mentioned above, prior downgrades can allow secret/untrusted information to directly influence the outer later of the data structure, but Condition 4 requires that all four trace prefixes be public-trusted equivalent, so any such downgrades must have the same influence across all executions. Condition 5 requires that some inputs result in prefixes that are public equivalent (or trusted equivalent for endorsement), while Condition 6 requires that other inputs result in prefixes that are distinguishable. Since all prefixes are public-trusted equivalent, this means there is an implicit downgrade inside a data structure, so the equivalent prefixes form an irrelevant input.

We can now relax our definition of robust declassification to only restrict the behavior of *relevant* inputs.

Definition 6.5 (Robust declassification). Let e be a program and let x and y be variables representing secret and untrusted inputs, respectively. We say that e *robustly declassifies* if, for all attackers $\mathcal{A}$ inducing high sets $\mathcal{U}$ and $\mathcal{S}$ (and $\mathcal{P} = \mathcal{L} \setminus \mathcal{S}$) and all values v_1, v_2, w_1, w_2, if

$$\langle e[x \mapsto v_i][y \mapsto w_j],\ v_i; w_j \rangle \longrightarrow\!\!\!\!\rightarrow^{*} \langle v_{ij},\ t^{ij} \rangle,$$

then $\left(\mathrm{rel}^{\leftarrow}_{\mathcal{A}, e}(w_1) \text{ and } t^{11} \approx^{\star}_{\mathcal{P}} t^{21}\right) \implies t^{12} \approx^{\star}_{\mathcal{P}} t^{22}$.

As NMIFC only restricts declassification of low-integrity data, endorsed data is free to influence future declassifications. As a result, we can only guarantee robust declassification in the absence of endorsements.

Theorem 6.6 (Robust declassification). *Given a program e, if $\Gamma, x:\tau_x, y:\tau_y; pc \vdash e : \tau$ and e contains no* `endorse` *expressions, then e robustly declassifies as defined in Definition 6.5.*

Note that prior definitions of robust declassification [11, 27] similarly prohibit endorsement and ignore pathological inputs, specifically nonterminating traces. Our irrelevant inputs are very different since NMIFC is strongly normalizing but admits complex data structures, but the need for some restriction is not new.

6.5 Transparent endorsement

We described in Section 2 how endorsing opaque writes can create security vulnerabilities. To formalize this intuition, we present *transparent endorsement*, a security condition that is dual to robust declassification. Instead of ensuring that untrusted information cannot meaningfully influence declassification, transparent endorsement guarantees that secret information cannot meaningfully influence endorsement. This guarantee ensures that secrets cannot influence the endorsement of an attacker's value—neither the value endorsed nor the existence of the endorsement itself.

As it is completely dual to robust declassification, we again appeal to the notion of irrelevant inputs, this time to rule out uninteresting secrets. The condition looks nearly identical, merely switching the roles of confidentiality and integrity. It therefore ensures that any choice of interesting secret provides an attacker with the maximum possible ability to influence endorsed values; no interesting secrets provide more power to attackers than others.

Definition 6.7 (Transparent endorsement). Let e be a program and let x and y be variables representing secret and untrusted inputs, respectively. We say that e *transparently endorses* if, for all attackers $\mathcal{A}$ inducing high sets $\mathcal{U}$ and $\mathcal{S}$ (and $\mathcal{T} = \mathcal{L} \setminus \mathcal{U}$) and all values v_1, v_2, w_1, w_2, if

$$\left\langle e[x \mapsto v_i][y \mapsto w_j],\, v_i; w_j \right\rangle \longrightarrow\!\!\!\!\rightarrow^* \left\langle v_{ij},\, t^{ij} \right\rangle,$$

then $\left(\mathrm{rel}^{\rightarrow}_{\mathcal{A},e}(v_1) \text{ and } t^{11} \approx^{\star}_{\mathcal{T}} t^{12}\right) \implies t^{21} \approx^{\star}_{\mathcal{T}} t^{22}$.

As in robust declassification, we can only guarantee transparent endorsement in the absence of declassification.

Theorem 6.8 (Transparent endorsement). *Given a program e, if $\Gamma, x:\tau_x, y:\tau_y; pc \vdash e : \tau$ and e contains no* `decl` *expressions, then e transparently endorses.*

6.6 Nonmalleable information flow

Robust declassification and transparent endorsement each restrict one type of downgrading, but as structured above, cannot be enforced in the presence of both declassification and endorsement. The key difficulty stems from the fact that previously declassified and endorsed data should be able to influence future declassifications and endorsements. However, any endorsement allows an attack to influence declassification, so varying the secret input can cause the traces to deviate for one attack and not another. Similarly, once a declassification has occurred, we can say little about the relation between trace pairs that fix a secret and vary an attack.

There is one condition that allows us to safely relate trace pairs even after a downgrade event: if the downgraded values are identical in both trace pairs. Even if a declassify or endorse could have caused the traces to deviate, if it did not, then this program is essentially the same as one that started with that value already downgraded and performed no downgrade. To capture this intuition, we define nonmalleable information flow in terms of trace prefixes that either do not deviate in public values when varying only the secret input or do not deviate in trusted values when varying only the untrusted input. This assumption may seem strong at first, but it exactly captures the intuition that downgraded data—but not secret/untrusted data—should be able to influence future downgrades. While two different endorsed attacks could influence a future declassification, if the attacks are similar enough to result in the same value being endorsed, they must influence the declassification *in the same way*.

Definition 6.9 (Nonmalleable information flow). Let e be a program and let x and y be variables representing secret and untrusted inputs, respectively. We say that e enforces *nonmalleable information flow* (NMIF) if the following holds for all attackers $\mathcal{A}$ inducing high sets $\mathcal{U}$ and $\mathcal{S}$. Let $\mathcal{T} = \mathcal{L} \setminus \mathcal{U}$, $\mathcal{P} = \mathcal{L} \setminus \mathcal{S}$ and $\mathcal{W} = \mathcal{T} \cap \mathcal{P}$. For all values v_1, v_2, w_1, and w_2, let

$$\left\langle e[x \mapsto v_i][y \mapsto w_j],\, v_i; w_j \right\rangle \longrightarrow\!\!\!\!\rightarrow^* \left\langle v_{ij},\, t^{ij} \right\rangle.$$

For all indices n_{ij} such that $t^{ij}_{n_{ij}} \not\approx_{\mathcal{W}} \bullet$

(1) If $t^{i1}_{..n_{i1}-1} \approx^{\star}_{\mathcal{T}} t^{i2}_{..n_{i2}-1}$ for $i = 1, 2$, then

$$\left(\mathrm{rel}^{\leftarrow}_{\mathcal{A},e}(w_1) \text{ and } t^{11}_{..n_{11}} \approx^{\star}_{\mathcal{P}} t^{21}_{..n_{21}}\right) \implies t^{12}_{..n_{12}} \approx^{\star}_{\mathcal{P}} t^{22}_{..n_{22}}.$$

(2) Similarly, if $t^{1j}_{..n_{1j}-1} \approx^{\star}_{\mathcal{P}} t^{2j}_{..n_{2j}-1}$ for $j = 1, 2$, then

$$\left(\mathrm{rel}^{\rightarrow}_{\mathcal{A},e}(v_1) \text{ and } t^{11}_{..n_{11}} \approx^{\star}_{\mathcal{T}} t^{12}_{..n_{12}}\right) \implies t^{21}_{..n_{21}} \approx^{\star}_{\mathcal{T}} t^{22}_{..n_{22}}.$$

Unlike the previous conditions, NMIFC enforces NMIF with no syntactic restrictions.

Note: this definition corrects a typographical error in the version published.

Theorem 6.10 (Nonmalleable information flow). *For any program e such that $\Gamma, x:\tau_x, y:\tau_y; pc \vdash e : \tau$, e enforces NMIF.*

We note that both Theorems 6.6 and 6.8 are directly implied by Theorem 6.10. For robust declassification, the syntactic prohibition on `endorse` directly enforces $t^{i1} \approx^{\star}_{\mathcal{T}} t^{i2}$ (for the entire trace), and the rest of case 1 is exactly that of Theorem 6.6. Similarly, the syntactic prohibition on `decl` enforces $t^{1j} \approx^{\star}_{\mathcal{P}} t^{2j}$, while the rest of case 2 is exactly Theorem 6.8.

7 NMIF AS 4-SAFETY

Clarkson and Schneider [13] define a *hyperproperty* as "a set of sets of infinite traces," and *hypersafety* to be a hyperproperty that can be characterized by a finite set of finite trace prefixes defining some "bad thing." That is, given any of these finite sets of trace prefixes it is impossible to extend those traces to satisfy the hyperproperty. It is therefore possible to show that a program satisfies a hypersafety property by proving that no set of finite trace prefixes emitted by the program fall into this set of "bad things." They further define a

k-safety hyperproperty (or *k-safety*) as a hypersafety property that limits the set of traces needed to identify a violation to size k.

Clarkson and Schneider note that noninterference provides an example of 2-safety. We demonstrate here that robust declassification, transparent endorsement, and nonmalleable information flow are all 4-safety properties.[4]

For a condition to be 2-safety, it must be possible to demonstrate a violation using only two finite traces. With noninterference, this demonstration is simple: if two traces with low-equivalent inputs are distinguishable by a low observer, the program is interfering.

Robust declassification, however, cannot be represented this way. It says that the program's confidentiality release events cannot be influence by untrusted inputs. If we could precisely identify the release events, this would allow us to specify robust declassification as a 2-safety property on those release events. If every pair of untrusted inputs results in the same trace of confidentiality release events, the program satisfies robust declassification. However, to identify confidentiality release events requires comparing traces with different secret inputs. A trace consists of a set of observable states, not a set of release events. Release events are identified by varying secrets; the robustness of releases is identified by varying untrusted input. Thus we need 4 traces to properly characterize robust declassification.

Both prior work [11] and our definition in Section 6.4 state robust declassification in terms of four traces, making it easy to convert to a 4-hyperproperty. That formulation cannot, however, be directly translated to 4-safety. It instead requires a statement about trace prefixes, which cannot be invalidated by extending traces.

Instead of simply reformulating Definition 6.5 with trace prefixes, we modify it using insights gained from the definition of NMIF. In particular, instead of a strict requirement that if a relevant attack results in public-equivalent trace prefixes then other attacks must as well, we relax this requirement to apply only when the trace prefixes are trusted-equivalent. As noted in Section 6.6, if we syntactically prohibit `endorse`—the only case in which we could enforce the previous definition—this trivially reduces to that definition. Without the syntactic restriction, however, the new condition is still enforceable.

For a given attacker $\mathcal{A}$ we can define a 4-safety property with respect to $\mathcal{A}$ (let $\mathcal{U}$, $\mathcal{S}$, $\mathcal{T}$, $\mathcal{P}$, and $\mathcal{W}$ be as in Definition 6.9).

$$
\begin{aligned}
\mathbf{RD}_{\mathcal{A}} \triangleq \Big\{ \mathbf{T} \subseteq \mathbb{T} \mid \mathbf{T} = \{t^{11}, t^{12}, t^{21}, t^{22}\} \\
\wedge\ t_1^{ij} \neq \bullet \wedge\ t_2^{ij} \neq \bullet \wedge\ t_1^{i1} = t_1^{i2} \wedge\ t_2^{1j} = t_2^{2j} \\
\implies \Big(\forall \{n_{ij}\} \subseteq \mathbb{N} : \big(t^{ij}_{n_{ij}} \not\approx_{\mathcal{W}} \bullet \wedge\ t^{i1}_{..n_{i1}-1} \approx^{\star}_{\mathcal{T}} t^{i2}_{..n_{i2}-1} \\
\wedge\ t^{11}_{..n_{11}} \approx^{\star}_{\mathcal{P}} t^{21}_{..n_{21}} \wedge\ t^{12}_{..n_{12}} \not\approx^{\star}_{\mathcal{P}} t^{22}_{..n_{22}} \big) \\
\implies t^{12}_{..n_{12}} \approx_{\mathcal{W}} t^{22}_{..n_{22}} \Big) \Big\}
\end{aligned}
$$

We then define robustness against all attackers as the intersection over all attackers: $\mathbf{RD} = \bigcap_{\mathcal{A}} \mathbf{RD}_{\mathcal{A}}$.

The above definition structurally combines Definition 6.4 with the first clause of Definition 6.9 to capture both the equivalence and the relevant-input statements of the original theorem. In the nested implication, if the first two clauses hold ($t^{ij}_{n_{ij}} \not\approx_{\mathcal{W}} \bullet$ and $t^{i1}_{..n_{i1}-1} \approx^{\star}_{\mathcal{T}} t^{i2}_{..n_{i2}-1}$), then one of three things must happen when fixing the attack and varying the secret: both trace pairs are equivalent, both trace pairs are non-equivalent, or the postcondition of the implication holds ($t^{12}_{..n_{12}} \approx_{\mathcal{W}} t^{22}_{..n_{22}}$). The first two satisfy the equivalency implication in Definition 6.9 while the third is exactly a demonstration that the first input is irrelevant.

[4] While NMIFC produces finite traces and hyperproperties are defined for infinite traces, we can easily extend NMIFC traces by stuttering $\bullet$ infinitely after termination.

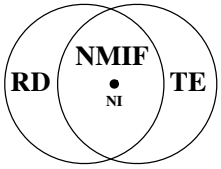

Figure 12: Relating 4-safety hyperproperties and noninterference.

Next we note that, while this does not strictly conform to the definition of robust declassification in Definition 6.5 which cannot be stated as a hypersafety property, **RD** is equivalent to Definition 6.5 for programs that do not perform endorsement. This endorse-free condition means that the equivalence clause $t^{i1}_{..n_{i1}-1} \approx^{\star}_{\mathcal{T}} t^{i2}_{..n_{i2}-1}$ will be true whenever the trace prefixes refer to the same point in execution. In particular, they can refer to the end of execution, which gives exactly the condition specified in the theorem.

As with every other result so far, the dual construction results in a 4-safety property **TE** representing transparent endorsement. Since **RD** captures the first clause of Definition 6.9, **TE** thus captures the second. This allows us to represent nonmalleable information flow as a 4-safety property very simply: $\mathbf{NMIF} = \mathbf{RD} \cap \mathbf{TE}$.

Figure 12 illustrates the relation between these hyperproperty definitions. Observe that the 2-safety hyperproperty **NI** for noninterference is contained in all three 4-safety hyperproperties. The insecure example programs of Section 2 are found in the left crescent, satisfying **RD** but not **NMIF**.

8 IMPLEMENTING NMIF

We have implemented the rules for nonmalleable information flow in context of Flame, a Haskell library and GHC [41] plugin. Flame provides data structures and compile-time checking of type-level acts-for constraints that are checked using a custom type-checker plugin. These constraints are used as the basis for encoding NMIFC as a shallowly-embedded domain-specific language (DSL). We have demonstrated that programs enforcing nonmalleable information flow can be built using this new DSL.

8.1 Information-flow monads in Flame

The DSL works by wrapping sensitive information in an abstract data type—a monad—that includes a principal type parameter representing the confidentiality and integrity of the information.

The Flame library tracks computation on protected information as a monadic effect and provides operations that constrain such computations to enforce information security. This effect is modeled using the `IFC` type class defined in Figure 13. The type class `IFC` is parameterized by two additional types, `n` in the `Labeled` type class and `e` in `Monad`. Instances of the `Labeled` type class enforce noninterference on pure computation—no downgrading or effects. The `e` parameter represents an effect we want to control.

```
class (Monad e, Labeled n) => IFC m e n where
  protect :: (pc ⊑ l) => a -> m e n pc l a

  use :: (l ⊑ l', pc ⊑ pc', l ⊑ pc', pc ⊑ pc'') =>
      m e n pc l a -> (a -> m e n pc' l' b)
                   -> m e n pc'' l' b

  runIFC :: m e n pc l a -> e (n l a)
```

Figure 13: Core information flow control operations in Flame.

```
class IFC m e n => NMIF m e n where
  declassify :: ( (C pc) ⊑ (C l)
                , (C l') ⊑ (C l) ⊔ Δ(I (l' ⊔ pc))
                , (I l') === (I l)) =>
              m e n pc l' a -> m e n pc l a

    endorse :: ( (I pc) ⊑ (I l)
                , (I l') ⊑ (I l) ⊔ ∇(C (l' ⊔ pc))
                , (C l') === (C l)) =>
              m e n pc l' a -> m e n pc l a
```

Figure 14: Nonmalleable information flow control in Flame.

```
recv :: (NMIF m e n, (I p) ⊑ ∇(C p)) =>
     n p a
     -> m e n (I (p ∧ q)) (p ∧ (I q)) a
recv v = endorse $ lift v

badrecv :: (NMIF m e n, (I p) ⊑ ∇(C p)) =>
        n (p ∧ C q) a
        -> m e n (I (p ∧ q)) (p ∧ q) a
badrecv v = endorse $ lift v {-REJECTED-}
```

Figure 15: Receive operations in `NMIF`. The secure `recv` is accepted, but the insecure `badrecv` is rejected.

For instance, many Flame libraries control effects in the `IO` monad, which is used for input, output, and mutable references.

The type `m e n pc l a` in Figure 13 associates a label `l` with the result of a computation of type `a`, as well as a program counter label `pc` that bounds the confidentiality and integrity of side effects for some effect `e`. Confidentiality and integrity projections are represented by type constructors `C` and `I`. The `protect` operator corresponds to monadic unit η (rule UnitM). Given any term, `protect` labels the term and lifts it into an `IFC` type where `pc ⊑ l`.

The `use` operation corresponds to a `bind` term in NMIFC. Its constraints implement the BindM typing rule. Given a protected value of type `m e n pc l a` and a function on a value of type `a` with return type `m e n pc' l' b`, `use` returns the result of applying the function, provided that `l ⊑ l'` and `(pc ⊔ l) ⊑ pc'`. Finally, `runIFC` executes a protected computation, which results in a labeled value of type `(n l a)` in the desired effect `e`.

We provide `NMIF`, which extends the `IFC` type class with `endorse` and `declassify` operations. The constraints on these operations implement the typing rules Endorse and Decl respectively.

We implemented the secure and insecure sealed-bid auction examples from Section 2.2 using `NMIF` operations, shown in Figure 15. As expected, the insecure `badrecv` is rejected by the compiler while the secure `recv` type checks.

```
authCheck :: Lbl MemoClient BasicAuthData
          -> NM IO (I MemoServer) (I MemoServer)
                   (BasicAuthResult Prin)
authCheck lauth =
 let lauth' = endorse $ lift lauth
     res = use lauth' $ \(BasicAuthData user guess) ->
           ebind user_db $ \db ->
           case Map.lookup user db of
            Nothing  -> protect Unauthorized
            Just pwd ->
              if guess == pwd then
               protect $ Authorized (Name user)
              else
               protect Unauthorized
  in declassify res
```

Figure 16: A nonmalleable password checker in Servant.

8.2 Nonmalleable HTTP Basic Authentication

To show the utility of NMIFC, we adapt a simple existing Haskell web application [21] based on the Servant [37] framework to run in Flame. The application allows users to create, fetch, and delete shared memos. Our version uses HTTP Basic Authentication and Flame's security mechanisms to restrict access to authorized users. We have deployed this application online at http://memo.flow.limited.

Figure 16 contains the function `authCheck`, which checks passwords in this application using the `NM` data type, which is an instance of the `NMIF` type class. The function takes a value containing the username and password guess of the authentication attempt, labeled with the confidentiality and integrity of an unauthenticated client, `MemoClient`. This value is endorsed to have the integrity of the server, `MemoServer`. This operation is safe since it only endorses information visible to the client. Next, the username is used to look up the correct password and compare it to the client's guess. If they match, then the user is authorized. The result of this comparison is secret, so before returning the result, it must be declassified.

Enforcing any form of information flow control on authentication mechanisms like `authCheck` provides more information security guarantees than traditional approaches. Unlike other approaches, nonmalleable information flow offers strong guarantees even when a computation endorses untrusted information. This example shows it is possible to construct applications that offer these guarantees.

9 RELATED WORK

Our efforts belong both within a significant body of work attempting to develop semantic security conditions that are more nuanced than noninterference, and within an overlapping body of work aiming to create expressive practical enforcement mechanisms for information flow control. Most prior work focuses on relaxing confidentiality restrictions; work permitting downgrading of integrity imposes relatively simple controls and lacks semantic security conditions that capture the concerns exemplified in Section 2.

Intransitive noninterference [29, 32, 34, 42] is an information flow condition that permits information to flow only between security levels (or *domains*) according to some (possibly intransitive) relation. It does not address the concerns of nonmalleability.

Decentralized information flow control (DIFC) [26] introduces the idea of mediating downgrading using access control [30]. However, the lack of robustness and transparency means downgrading can still be exploited in these systems (e.g., [16, 22, 25, 48]).

Robust declassification and qualified robustness have been explored in DIFC systems as a way to constrain the adversary's influence on declassification [4–6, 12, 27, 46, 47]. While transparent endorsement can be viewed as an integrity counterpart to robust declassification, this idea is not present in prior work.

Sabelfeld and Sands provide a clarifying taxonomy for much prior work on declassification [36], introducing various dimensions along which declassification mechanisms operate. They categorize robust declassification as lying on the "who" dimension. However, they do not explicitly consider endorsement mechanisms. Regardless of the taxonomic category, transparent endorsement and nonmalleable information flow also seem to lie on the same dimension as robust declassification, since they take into account influences on the information that is downgraded.

Label algebras [24] provide an abstract characterization of several DIFC systems. However, they do not address the restrictions on downgrading imposed by nonmalleable information flow.

The Aura language [20] uses information flow policies to constrain authorization and endorsement. However, it does not address the malleability of endorsement. Rx [40] represents information flow control policies in terms of dynamic *roles* [18]. Adding new principals to these roles corresponds to declassification and endorsement since new flows may occur. Rx constrains updates to roles similarly to previous type systems that enforce robust declassification and qualified robustness but does not prevent opaque endorsements.

Relational Hoare Type Theory [28] (RHTT) offers a powerful and precise way to specify security conditions that are 2-hyperproperties, such as noninterference. Cartesian Hoare logic [38] (CHL) extends standard Hoare logic to reason about k-safety properties of relational traces (the input/output pairs of a program). Since nonmalleable information flow, robust declassification, and transparent endorsement are all 4-safety properties that cannot be fully expressed with relational traces, neither RHTT nor CHL can characterize them properly.

Haskell's type system has been attractive target for embedding information flow checking [9, 23, 39]. Much prior work has focused on dynamic information flow control. LIO [39] requires computation on protected information to occur in the `LIO` monad, which tracks the confidentiality and integrity of information accessed ("unlabeled") by the computation. HLIO [9] explores hybrid static and dynamic enforcement. Flame enforces information flow control statically, and the `NMIF` type class enforces nonmalleable IFC statically as well. The static component of HLIO enforces solely via the Haskell type system (and existing general-purpose extensions), but Flame—and by extension, `NMIF`—uses custom constraints based on the FLAM algebra which are processed by a GHC type checker plugin. Extending the type checker to reason about FLAM constraints significantly improves programmability over pure-Haskell approaches like HLIO.

10 CONCLUSION

Downgrading mechanisms like declassification and endorsement make information flow mechanisms sufficiently flexible and expressive for real programs. However, we have shown that previous notions of information-flow security missed the dangers endorsing confidential information. We therefore introduced transparent endorsement as a security property that rules out such influences and showed that it is dual to robust declassification. Robust declassification and transparent endorsement are both consequences of a stronger property, nonmalleable information flow, and we have formulated all three as 4-safety properties. We have shown how to provably enforce these security properties in an efficient, compositional way using a security type system. Based on our Haskell implementation, these security conditions and enforcement mechanism appear to be practical, supporting the secure construction of programs with complex security requirements.

While security-typed languages are not yet mainstream, information flow control, sometimes in the guise of taint tracking, has become popular as a way to detect and control real-world vulnerabilities (e.g., [17]). Just as the program analyses used are approximations of previous security type systems targeting noninterference, it is reasonable to expect the NMIFC type system to be a useful guide for other analyses and enforcement mechanisms.

ACKNOWLEDGMENTS

Many people helped us with this work. Martín Abadi posed a provocative question about dualities. Pablo Buiras helped develop the memo example. David Naumann pointed out work on k-safety. Tom Magrino, Yizhou Zhang, and the anonymous reviewers gave us useful feedback on the paper.

Funding for this work was provided by NSF grants 1513797 and 1524052, and by a gift from Google. Any opinions, findings, conclusions, or recommendations expressed here are those of the author(s) and do not necessarily reflect those of these sponsors.

REFERENCES

[1] Martín Abadi. 2006. Access Control in a Core Calculus of Dependency. In *11th ACM SIGPLAN Int'l Conf. on Functional Programming*. ACM, New York, NY, USA, 263–273.

[2] Martín Abadi. 2008. Variations in Access Control Logic. In *Deontic Logic in Computer Science*, Ron van der Meyden and Leendert van der Torre (Eds.). Lecture Notes in Computer Science, Vol. 5076. Springer Berlin Heidelberg, 96–109.

[3] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. 1999. A Core Calculus of Dependency. In *26th ACM Symp. on Principles of Programming Languages (POPL)*. 147–160.

[4] Owen Arden, Jed Liu, and Andrew C. Myers. 2015. Flow-Limited Authorization. In *28th IEEE Symp. on Computer Security Foundations (CSF)*. 569–583.

[5] Owen Arden and Andrew C. Myers. 2016. A Calculus for Flow-Limited Authorization. In *29th IEEE Symp. on Computer Security Foundations (CSF)*. 135–147.

[6] Aslan Askarov and Andrew C. Myers. 2011. Attacker Control and Impact for Confidentiality and Integrity. *Logical Methods in Computer Science* 7, 3 (Sept. 2011).

[7] K. J. Biba. 1977. *Integrity Considerations for Secure Computer Systems.* Technical Report ESD-TR-76-372. USAF Electronic Systems Division, Bedford, MA. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[8] Niklas Broberg and David Sands. 2010. Paralocks—Role-Based Information Flow Control and Beyond. In *37th ACM Symp. on Principles of Programming Languages (POPL)*.

[9] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell. In *20th ACM SIGPLAN Int'l Conf. on Functional Programming*. ACM, 289–301.

[10] Ethan Cecchetti, Andrew C. Myers, and Owen Arden. 2017. *Nonmalleable Information Flow Control: Technical Report.* Technical Report. Cornell University Computing and Information Science. https://arxiv.org/abs/1708.08596.

[11] Stephen Chong and Andrew C. Myers. 2006. Decentralized Robustness. In *19th IEEE Computer Security Foundations Workshop (CSFW)*. 242–253.

[12] Stephen Chong and Andrew C. Myers. 2008. End-to-End Enforcement of Erasure and Declassification. In *IEEE Symp. on Computer Security Foundations (CSF)*. 98–111.

[13] Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *IEEE Symp. on Computer Security Foundations (CSF)*. 51–65.

[14] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Comm. of the ACM* 19, 5 (1976), 236–243.

[15] Danny Dolev, Cynthia Dwork, and Moni Naor. 2003. Nonmalleable Cryptography. *SIAM Rev.* 45, 4 (2003), 727–784.

[16] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. 2005. Labels and Event Processes in the Asbestos Operating System. In *20th ACM Symp. on Operating System Principles (SOSP)*.

[17] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. 2014. Collaborative Verification of Information Flow for a High-Assurance App Store. In *21st ACM Conf. on Computer and Communications Security (CCS)*. 1092–1104.

[18] David Ferraiolo and Richard Kuhn. 1992. Role-Based Access Controls. In *15th National Computer Security Conference*.

[19] Joseph A. Goguen and Jose Meseguer. 1982. Security Policies and Security Models. In *IEEE Symp. on Security and Privacy*. 11–20.

[20] Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. 2008. Aura: A Programming Language for Authorization and Audit. In *13th ACM SIGPLAN Int'l Conf. on Functional Programming*.

[21] krdlab. 2014. Haskell Servant Example. https://github.com/krdlab/examples. (Dec. 2014).

[22] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information Flow Control for Standard OS Abstractions. In *21st ACM Symp. on Operating System Principles (SOSP)*.

[23] Peng Li and Steve Zdancewic. 2006. Encoding information flow in Haskell. In *19th IEEE Computer Security Foundations Workshop (CSFW)*.

[24] Benoît Montagu, Benjamin C. Pierce, and Randy Pollack. 2013. A Theory of Information-Flow Labels. In *26th IEEE Symp. on Computer Security Foundations (CSF)*. 3–17.

[25] Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *26th ACM Symp. on Principles of Programming Languages (POPL)*. 228–241.

[26] Andrew C. Myers and Barbara Liskov. 2000. Protecting Privacy using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology* 9, 4 (Oct. 2000), 410–442.

[27] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. 2006. Enforcing Robust Declassification and Qualified Robustness. *Journal of Computer Security* 14, 2 (2006), 157–196.

[28] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. 2011. Verification of Information Flow and Access Control Policies with Dependent Types. In *IEEE Symp. on Security and Privacy*. 165–179.

[29] Sylvan Pinsky. 1995. Absorbing Covers and Intransitive Non-Interference. In *IEEE Symp. on Security and Privacy*. 102–113.

[30] François Pottier and Sylvain Conchon. 2000. Information Flow Inference for Free. In *5th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP '00)*. 46–57.

[31] François Pottier and Vincent Simonet. 2003. Information Flow Inference for ML. *ACM Trans. on Programming Languages and Systems* 25, 1 (Jan. 2003).

[32] A. W. Roscoe and M. H. Goldsmith. 1999. What is Intransitive Noninterference?. In *12th IEEE Computer Security Foundations Workshop (CSFW)*. 228–238.

[33] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. 2009. Laminar: Practical Fine-Grained Decentralized Information Flow Control. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*.

[34] John Rushby. 1992. *Noninterference, transitivity and channel-control security policies.* Technical Report CSL-92-02. SRI.

[35] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (Jan. 2003), 5–19.

[36] Andrei Sabelfeld and David Sands. 2005. Dimensions and Principles of Declassification. In *18th IEEE Computer Security Foundations Workshop (CSFW)*. 255–269.

[37] Servant Contributors. 2016. Servant – A Type-Level Web DSL. http://haskell-servant.readthedocs.io/. (2016).

[38] Marcelo Sousa and Isil Dillig. 2016. Cartesian Hoare logic for verifying k-safety properties. In *SIGPLAN Notices*, Vol. 51. ACM, 57–69.

[39] Deian Stefan, Amit Levy, Alejandro Russo, and David Mazières. 2014. Building Secure Systems with LIO. In *Haskell Symposium*. ACM SIGPLAN.

[40] Nikhil Swamy, Michael Hicks, Stephen Tse, and Steve Zdancewic. 2006. Managing Policy Updates in Security-Typed Languages. In *19th IEEE Computer Security Foundations Workshop (CSFW)*. 202–216.

[41] The Glasgow Haskell Compiler 2016. The Glasgow Haskell Compiler. (2016). https://www.haskell.org/ghc/.

[42] Ron van der Meyden. 2007. What, Indeed, Is Intransitive Noninterference?. In *12th European Symposium on Research in Computer Security (ESORICS)*. 235–250.

[43] Lucas Waye, Pablo Buiras, Dan King, Stephen Chong, and Alejandro Russo. 2015. It's My Privilege: Controlling Downgrading in DC-Labels. In *Proceedings of the 11th International Workshop on Security and Trust Management*.

[44] J. Todd Wittbold and Dale M. Johnson. 1990. Information Flow in Nondeterministic Systems. In *IEEE Symp. on Security and Privacy*. 144–161.

[45] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38–94.

[46] Steve Zdancewic and Andrew C. Myers. 2001. Robust Declassification. In *14th IEEE Computer Security Foundations Workshop (CSFW)*. 15–23.

[47] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. 2002. Secure Program Partitioning. *ACM Trans. on Computer Systems* 20, 3 (Aug. 2002), 283–328.

[48] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making Information Flow Explicit in HiStar. In *7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*. 263–278.

[49] Lantian Zheng and Andrew C. Myers. 2007. Dynamic Security Labels and Static Information Flow Control. *International Journal of Information Security* 6, 2–3 (March 2007).

A FULL NMIFC

We present the full syntax, semantics, and typing rules for NMIFC in Figures 17, 18, and 20, respectively. This is a straightforward extension of the core language presented in Section 5. We note that polymorphic terms specify a pc just as λ terms. This is because they contain arbitrary expressions which could produce arbitrary effects, so we must constrain the context that can execute those effects.

Figure 21 presents the full set of derivation rules for the acts-for (delegation) relation $p \succcurlyeq q$.

A.1 Label tracking with brackets

In order to simply proofs of hyperproperties requiring 2 and 4 traces, we introduce a new bracket syntax to track secret and untrusted data. These brackets are inspired by those used by Pottier and Simonet [31] to prove their FlowCaml type system enforced noninterference. Their brackets served two purposes simultaneously. First they allow a single execution of a bracketed program to faithfully model two executions of a non-bracketed program. Second, the brackets track secret/untrusted information through

$n \in \mathcal{N}$ (atomic principals)
$x \in \mathcal{V}$ (variable names)

$$
\begin{array}{rcl}
p, \ell, pc & ::= & n \mid \top \mid \bot \mid p^{\pi} \mid p \wedge p \mid p \vee p \mid p \sqcup p \mid p \sqcap p \\
\tau & ::= & \texttt{unit} \mid X \mid (\tau + \tau) \mid (\tau \times \tau) \\
& \mid & \tau \xrightarrow{pc} \tau \mid \forall X[pc].\, \tau \mid \ell \ \texttt{says}\ \tau \\
v & ::= & () \mid \texttt{inj}_i\ v \mid \langle v, v \rangle \mid (\overline{\eta}_\ell\ v) \\
& \mid & \lambda(x\!:\!\tau)[pc].\, e \mid \Lambda X[pc].\, e \\
e & ::= & x \mid v \mid e\ e \mid e\ \tau \mid \langle e, e \rangle \mid (\eta_\ell\ e) \\
& \mid & \texttt{proj}_i\ e \mid \texttt{inj}_i\ e \mid \texttt{bind}\ x = e\ \texttt{in}\ e \\
& \mid & \texttt{case}\ e\ \texttt{of}\ \texttt{inj}_1(x).e \mid \texttt{inj}_2(x).e \\
& \mid & \texttt{decl}\ e\ \texttt{to}\ \ell \mid \texttt{endorse}\ e\ \texttt{to}\ \ell
\end{array}
$$

Figure 17: Full NMIFC syntax.

$\boxed{e \longrightarrow e'}$

[E-App] $(\lambda(x:\tau)[pc].\ e)\ v \longrightarrow e[x \mapsto v]$

[E-TApp] $(\Lambda X[pc].\ e)\ \tau \longrightarrow e[X \mapsto \tau]$

[E-UnPair] $\texttt{proj}_i\ \langle v_1, v_2\rangle \longrightarrow v_i$

[E-Case] $(\texttt{case}\ (\texttt{inj}_i\ v)\ \texttt{of}\ \texttt{inj}_1(x).e_1 \mid \texttt{inj}_2(x).e_2) \longrightarrow e_i[x \mapsto v]$

[E-BindM] $\texttt{bind}\ x = (\overline{\eta}_\ell\ v)\ \texttt{in}\ e \longrightarrow e[x \mapsto v]$

$\boxed{\langle e,\ t\rangle \longrightarrow\!\!\!\!\rightarrow \langle e',\ t'\rangle}$

[E-Step] $$\frac{e \longrightarrow e'}{\langle e,\ t\rangle \longrightarrow\!\!\!\!\rightarrow \langle e',\ t;\bullet\rangle}$$

[E-UnitM] $\langle(\eta_\ell\ v),\ t\rangle \longrightarrow\!\!\!\!\rightarrow \langle(\overline{\eta}_\ell\ v),\ t;(\overline{\eta}_\ell\ v)\rangle$

[E-Decl] $\langle\texttt{decl}\ (\overline{\eta}_{\ell'}\ v)\ \texttt{to}\ \ell,\ t\rangle \longrightarrow\!\!\!\!\rightarrow \langle(\overline{\eta}_\ell\ v),\ t;(\downarrow^{\rightarrow}_{\ell'},\ \overline{\eta}_\ell\ v)\rangle$

[E-Endorse] $\langle\texttt{endorse}\ (\overline{\eta}_{\ell'}\ v)\ \texttt{to}\ \ell,\ t\rangle \longrightarrow\!\!\!\!\rightarrow \langle(\overline{\eta}_\ell\ v),\ t;(\downarrow^{\leftarrow}_{\ell'},\ \overline{\eta}_\ell\ v)\rangle$

[E-Eval] $$\frac{\langle e,\ t\rangle \longrightarrow\!\!\!\!\rightarrow \langle e',\ t'\rangle}{\langle E[e],\ t\rangle \longrightarrow\!\!\!\!\rightarrow \langle E[e'],\ t'\rangle}$$

Evaluation context

$$\begin{array}{rcl} E & ::= & [\cdot] \mid E\ e \mid v\ E \mid E\ \tau \mid \langle E, e\rangle \mid \langle v, E\rangle \mid (\eta_\ell\ E) \\ & \mid & \texttt{proj}_i\ E \mid \texttt{inj}_i\ E \mid \texttt{bind}\ x = E\ \texttt{in}\ e \\ & \mid & \texttt{case}\ E\ \texttt{of}\ \texttt{inj}_1(x).e \mid \texttt{inj}_2(x).e \\ & \mid & \texttt{decl}\ E\ \texttt{to}\ \ell \mid \texttt{endorse}\ E\ \texttt{to}\ \ell \end{array}$$

Figure 18: Full NMIFC operational semantics.

$\boxed{\vdash \ell \sqsubseteq \tau}$

[P-Unit] $\vdash \ell \sqsubseteq \texttt{unit}$ [P-Lbl] $$\frac{\ell' \sqsubseteq \ell}{\vdash \ell' \sqsubseteq \ell\ \texttt{says}\ \tau}$$

[P-Pair] $$\frac{\vdash \ell \sqsubseteq \tau_1 \qquad \vdash \ell \sqsubseteq \tau_2}{\vdash \ell \sqsubseteq (\tau_1 \times \tau_2)}$$

$\boxed{\vdash \tau \sqsubseteq \mathcal{H}}$

[P-Set] $$\frac{H \in \mathcal{H} \qquad \vdash H \sqsubseteq \tau}{\vdash \tau\ \text{prot}\ \mathcal{H}}\ \mathcal{H}\text{ is upward closed}$$

Figure 19: Type protection levels.

$\boxed{\Gamma; pc \vdash e : \tau}$

[Var] $\Gamma, x:\tau, \Gamma'; pc \vdash x : \tau$ [Unit] $\Gamma; pc \vdash () : \texttt{unit}$

[Lam] $$\frac{\Gamma, x:\tau_1; pc' \vdash e : \tau_2}{\Gamma; pc \vdash \lambda(x:\tau_1)[pc'].\ e : \tau_1 \xrightarrow{pc'} \tau_2}$$ [App] $$\frac{\Gamma; pc \vdash e_1 : \tau' \xrightarrow{pc'} \tau \qquad \Gamma; pc \vdash e_2 : \tau' \qquad pc \sqsubseteq pc'}{\Gamma; pc \vdash e_1\ e_2 : \tau}$$

[TLam] $$\frac{\Gamma, X; pc' \vdash e : \tau}{\Gamma; pc \vdash \Lambda X[pc'].\ e : \forall X[pc'].\ \tau}$$

[TApp] $$\frac{\Gamma; pc \vdash e : \forall X[pc'].\ \tau \qquad pc \sqsubseteq pc'}{\Gamma; pc \vdash (e\ \tau') : \tau[X \mapsto \tau']}\ \tau'\text{ is well-formed in }\Gamma$$

[Pair] $$\frac{\Gamma; pc \vdash e_1 : \tau_1 \qquad \Gamma; pc \vdash e_2 : \tau_2}{\Gamma; pc \vdash \langle e_1, e_2\rangle : (\tau_1 \times \tau_2)}$$ [UnPair] $$\frac{\Gamma; pc \vdash e : (\tau_1 \times \tau_2)}{\Gamma; pc \vdash \texttt{proj}_i\ e : \tau_i}$$

[Inj] $$\frac{\Gamma; pc \vdash e : \tau_i}{\Gamma; pc \vdash \texttt{inj}_i\ e : (\tau_1 + \tau_2)}$$

[Case] $$\frac{\Gamma; pc \vdash e : (\tau_1 + \tau_2) \qquad \vdash pc \sqsubseteq \tau \qquad \Gamma, x:\tau_1; pc \vdash e_1 : \tau \qquad \Gamma, x:\tau_2; pc \vdash e_2 : \tau}{\Gamma; pc \vdash \texttt{case}\ e\ \texttt{of}\ \texttt{inj}_1(x).e_1 \mid \texttt{inj}_2(x).e_2 : \tau}$$

[UnitM] $$\frac{\Gamma; pc \vdash e : \tau \qquad pc \sqsubseteq \ell}{\Gamma; pc \vdash (\eta_\ell\ e) : \ell\ \texttt{says}\ \tau}$$ [VUnitM] $$\frac{\Gamma; pc \vdash v : \tau}{\Gamma; pc \vdash (\overline{\eta}_\ell\ v) : \ell\ \texttt{says}\ \tau}$$

[BindM] $$\frac{\Gamma; pc \vdash e : \ell\ \texttt{says}\ \tau' \qquad \vdash \ell \sqsubseteq \tau \qquad \Gamma, x:\tau'; pc \sqcup \ell \vdash e' : \tau}{\Gamma; pc \vdash \texttt{bind}\ x = e\ \texttt{in}\ e' : \tau}$$

[Decl] $$\frac{\Gamma; pc \vdash e : \ell'\ \texttt{says}\ \tau \qquad \ell'^{\leftarrow} = \ell^{\leftarrow} \qquad pc \sqsubseteq \ell \qquad \ell'^{\rightarrow} \sqsubseteq \ell^{\rightarrow} \sqcup \Delta((\ell' \sqcup pc)^{\leftarrow})}{\Gamma; pc \vdash \texttt{decl}\ e\ \texttt{to}\ \ell : \ell\ \texttt{says}\ \tau}$$

[Endorse] $$\frac{\Gamma; pc \vdash e : \ell'\ \texttt{says}\ \tau \qquad \ell'^{\rightarrow} = \ell^{\rightarrow} \qquad pc \sqsubseteq \ell \qquad \ell'^{\leftarrow} \sqsubseteq \ell^{\leftarrow} \sqcup \nabla((\ell' \sqcup pc)^{\rightarrow})}{\Gamma; pc \vdash \texttt{endorse}\ e\ \texttt{to}\ \ell : \ell\ \texttt{says}\ \tau}$$

Figure 20: Typing rules for full NMIFC language.

$\boxed{p \succcurlyeq q}$

[Bot] $p \succcurlyeq \bot$ [Top] $\top \succcurlyeq p$ [Refl] $p \succcurlyeq p$ [Proj] $$\frac{p \succcurlyeq q}{p^\pi \succcurlyeq q^\pi}$$

[ProjR] $p \succcurlyeq p^\pi$ [ConjL] $$\frac{p_i \succcurlyeq q \qquad i \in \{1,2\}}{p_1 \wedge p_2 \succcurlyeq q}$$ [ConjR] $$\frac{p \succcurlyeq q_1 \qquad p \succcurlyeq q_2}{p \succcurlyeq q_1 \wedge q_2}$$

[DisL] $$\frac{p_1 \succcurlyeq q \qquad p_2 \succcurlyeq q}{p_1 \vee p_2 \succcurlyeq q}$$ [DisR] $$\frac{p \succcurlyeq q_i \qquad i \in \{1,2\}}{p \succcurlyeq q_1 \vee q_2}$$ [Trans] $$\frac{p \succcurlyeq q \qquad q \succcurlyeq r}{p \succcurlyeq r}$$

Figure 21: Principal lattice rules

execution of the program, thereby making it easy to verify that it did not interfere with public/trusted information simply by proving that brackets could not be syntactically present in such values. Since noninterference only requires examining pairs of traces, these purposes complement each other well; if the two executions vary only on high inputs, then low outputs cannot contain brackets.

While this technique is very effective to prove noninterference, nonmalleable information flow provides security guarantees even in the presence of both declassification and endorsement. As a result, we need to track secret/untrusted information even through downgrading events that can cause traces to differ arbitrarily. To accomplish this goal, we use brackets that serve only the second purpose: they track restricted information but not multiple executions.

As in previous formalizations, NMIFC's brackets are defined with respect to a notion of "high" labels, in this case a high set. The high set restricts the type of the expression inside the bracket as well as the *pc* at which it must type, thereby restricting the effects it can create. For the more complex theorems we must track data with multiple different high labels within the same program execution, so we parameterize the brackets themselves with the

Syntax extensions

$$v ::= \cdots \mid (\!| v |\!)_{\mathcal{H}}$$
$$e ::= \cdots \mid (\!| e |\!)_{\mathcal{H}}$$

New contexts

$$E ::= \cdots \mid (\!| E |\!)_{\mathcal{H}}$$
$$B ::= \texttt{proj}_i\,[\cdot] \mid \texttt{bind}\ x = [\cdot]\ \texttt{in}\ e$$

Evaluation extensions

[B-Expand] $$B[(\!| v |\!)_{\mathcal{H}}] \longrightarrow (\!| B[v] |\!)_{\mathcal{H}}$$

[B-DeclL] $$\frac{\ell \notin \mathcal{H}}{\texttt{decl}\ (\!| v |\!)_{\mathcal{H}}\ \texttt{to}\ \ell \longrightarrow \texttt{decl}\ v\ \texttt{to}\ \ell}$$

[B-DeclH] $$\frac{\ell \in \mathcal{H}}{\texttt{decl}\ (\!| v |\!)_{\mathcal{H}}\ \texttt{to}\ \ell \longrightarrow (\!|\texttt{decl}\ v\ \texttt{to}\ \ell |\!)_{\mathcal{H}}}$$

[B-EndorseL] $$\frac{\ell \notin \mathcal{H}}{\texttt{endorse}\ (\!| v |\!)_{\mathcal{H}}\ \texttt{to}\ \ell \longrightarrow \texttt{endorse}\ v\ \texttt{to}\ \ell}$$

[B-EndorseH] $$\frac{\ell \in \mathcal{H}}{\texttt{endorse}\ (\!| v |\!)_{\mathcal{H}}\ \texttt{to}\ \ell \longrightarrow (\!|\texttt{endorse}\ v\ \texttt{to}\ \ell |\!)_{\mathcal{H}}}$$

Typing extensions

[Bracket] $$\frac{\Gamma; pc' \vdash e : \tau \qquad pc \sqsubseteq pc' \qquad pc' \in \mathcal{H} \qquad \vdash \tau\ \text{prot}\ \mathcal{H}}{\Gamma; pc \vdash (\!| e |\!)_{\mathcal{H}} : \tau}\ \mathcal{H}\ \text{is upward closed}$$

Bracket projection

$$\lfloor e \rfloor = \begin{cases} \lfloor e' \rfloor & \text{if } e = (\!| e' |\!)_{\mathcal{H}} \\ \text{recursively project all sub-expressions} & \text{otherwise} \end{cases}$$

Figure 22: NMIFC language extensions.

high set. We present the extended syntax, semantics, and typing rules in Figure 22.

B ATTACKER PROPERTIES

Recall that we defined an attacker as a set of principals $\mathcal{A} = \{\ell \in \mathcal{L} \mid n_1 \wedge \cdots \wedge n_k \succcurlyeq \ell\}$ for some non-empty finite set of atomic principals $\{n_1, \ldots, n_k\} \subseteq \mathcal{N}$.

Definition B.1 (Attacker properties). Let $\mathcal{A}$ be an attacker and let $\mathcal{A}^\pi = \{p \in \mathcal{L} \mid \exists q \in \mathcal{L}$ such that $p^\pi \wedge q^{\pi'} \in \mathcal{A}\}$. The following properties hold:

1. for all $a_1, a_2 \in \mathcal{A}^\pi$, $a_1 \wedge a_2 \in \mathcal{A}^\pi$ (Attacking principals may collude)
2. for all $a \in \mathcal{A}^\pi$ and $b \in \mathcal{L}$, $a \vee b \in \mathcal{A}^\pi$ (Attackers may attenuate their power)
3. for all $b_1, b_2 \notin \mathcal{A}^\pi$, $b_1 \vee b_2 \notin \mathcal{A}^\pi$ (Combining public information in a public context yields public information and combining trusted information in a trusted context yields trusted information)
4. for all $a \in \mathcal{L}$ and $b \notin \mathcal{A}^\pi$, $a \wedge b \notin \mathcal{A}^\pi$ (Attackers cannot compromise policies requiring permission from non-attacking principals)
5. for all $a \in \mathcal{A}$, $\nabla(a^{\rightarrow}) \wedge \Delta(a^{\leftarrow}) \in \mathcal{A}$. (Attackers have the same power in confidentiality and integrity)

The theorems proved in this paper hold for any attacker satisfying these properties, so for generality we can take the properties in Definition B.1 as defining an attacker.

We now prove that our original definition of an attacker satisfies Definition B.1.

Proof. Conditions 1 and 2 of Definition B.1 follow directly from the definition of $\mathcal{A}$ and ConjR and DisR, respectively. Condition 5 holds by the symmetry of the lattice.

Since we are only examining one of confidentiality and integrity at a time, for the following conditions we assume without loss of generality that all principals in each expression have only the π projection and the other component is $\bot$. In particular, this means we can assume Proj and ProjR are not used in any derivation, and any application of the conjunction or disjunction derivation rules split in a meaningful way with respect to the π projection (i.e., neither principal in the side being divided is $\top$ or $\bot$).

We now show Condition 4 holds by contradiction. Assume $a \in \mathcal{L}$ and $b \notin \mathcal{A}^\pi$, but $a \wedge b \in \mathcal{A}^\pi$. This means $(n_1 \wedge \cdots \wedge n_k)^\pi \succcurlyeq a \wedge b$. We prove by induction on k that $a, b \in \mathcal{A}^\pi$. If $k = 1$, then the only possible rule to derive this result is ConjL and we are finished. If $k > 1$, then the derivation of this relation must be due to either ConjL or ConjR. If it is due to ConjR, then this again achieves the desired contradiction. If it is due to ConjL, then the same statement holds for a subset of the atomic principals $n'_1, \ldots, n'_{k'}$, where $k' < k$, so by induction, $(n'_1 \wedge \cdots \wedge n'_{k'})^\pi \succcurlyeq b^\pi$, and by Trans, $(n_1 \wedge \ldots \wedge n_k)^\pi \succcurlyeq b^\pi$ which also contradicts our assumption.

Finally, we also show Condition 3 holds by contradiction. We assume $b_1, b_2 \notin \mathcal{A}^\pi$ but $b_1 \vee b_2 \in \mathcal{A}^\pi$ and again prove a contradiction by induction on k. If $k = 1$, then the derivation showing $n_1^\pi \succcurlyeq (b_1 \vee b_2)^\pi$ must end with DisR which contradicts the assumption that $b_1, b_2 \notin \mathcal{A}^\pi$. If $k > 1$, the derivation either ends with DisR, resulting in the same contradiction, or with ConjL. In this second case, the same argument as above holds: there is a strict subset of the principals $n_1, \ldots, n_k$ that act for either b_1 or b_2 and thus by Trans we acquire the desired contradiction. □

C GENERALIZATION

Definition 6.9 (and correspondingly Theorem 6.10) might appear relatively narrow; they only speak directly to programs with a single untrusted value and a single secret value. However, because the language has first-class functions and pair types, the theorem as stated is equivalent to one that allows insertion of secret and untrusted code into multiple points in the program, as long as that code types in an appropriately restrictive *pc*.

To define this formally, we first need a means to allow for insertion of arbitrary code. We follow previous work [27] by extending the language to include *holes*. A program expression may contain an ordered set of holes. These holes may be replaced with arbitrary expressions, under restrictions requiring that the holes be treated as sufficiently secret or untrusted. Specifically, the type system is extended with the following rule:

[Hole] $$\frac{pc \in \mathcal{H} \qquad \vdash \tau\ \text{prot}\ \mathcal{H}}{\Gamma; pc \vdash [\bullet]_{\mathcal{H}} : \tau}\ \mathcal{H}\ \text{is a high set}$$

Using this definition, we can state NMIF in a more traditional form.

Definition C.1 (General NMIF). We say that a program $e[\vec{\bullet}]_{\mathcal{H}}$ enforces *general NMIF* if the following holds for all attackers $\mathcal{A}$ inducing high sets $\mathcal{U}$ and $\mathcal{S}$. Let $\mathcal{T} = \mathcal{L} \setminus \mathcal{U}$, $\mathcal{P} = \mathcal{L} \setminus \mathcal{S}$ and $\mathcal{W} = \mathcal{T} \cap \mathcal{S}$. If $\mathcal{H} \subseteq \mathcal{U}$, then for all values v_1, v_2 and all attacks $\vec{a}_1$ and $\vec{a}_2$, let

$$\langle e[\vec{a}_i]_{\mathcal{H}}[\vec{x} \mapsto \vec{v}_i],\, \vec{v}_i\rangle \longrightarrow\!\!\!\!\rightarrow^{*} \left\langle v_{ij},\, t^{ij}\right\rangle.$$

For all indices n_{ij} such that $t^{ij}_{n_{ij}} \not\approx_{\mathcal{W}} \bullet$

(1) If $t^{i1}_{\ldots n_{i1}-1} \approx^{\star}_{\mathcal{T}} t^{i2}_{\ldots n_{i2}-1}$ for $i = 1, 2$, then

$$\left(\mathrm{rel}^{\leftarrow}_{\mathcal{A},e}(w_1) \text{ and } t^{11}_{\ldots n_{11}} \approx^{\star}_{\mathcal{P}} t^{21}_{\ldots n_{21}}\right) \implies t^{12}_{\ldots n_{12}} \approx^{\star}_{\mathcal{P}} t^{22}_{\ldots n_{22}}.$$

(2) Similarly, if $t^{1j}_{\ldots n_{1j}-1} \approx^{\star}_{\mathcal{P}} t^{2j}_{\ldots n_{2j}-1}$ for $j = 1, 2$, then

$$\left(\mathrm{rel}^{\rightarrow}_{\mathcal{A},e}(v_1) \text{ and } t^{11}_{\ldots n_{11}} \approx^{\star}_{\mathcal{T}} t^{12}_{\ldots n_{12}}\right) \implies t^{21}_{\ldots n_{21}} \approx^{\star}_{\mathcal{T}} t^{22}_{\ldots n_{22}}.$$

For NMIFC, this definition is equivalent to Definition 6.9. We prove this fact to prove the following theorem.

Theorem C.2 (General NMIF). *Given a program $e[\vec{\bullet}]_{\mathcal{H}}$ such that $\Gamma, \vec{x}:\vec{\tau}; pc \vdash e[\vec{\bullet}]_{\mathcal{H}} : \tau'$, then $e[\vec{\bullet}]_{\mathcal{H}}$ enforces general NMIF.*

Proof. We prove this by reducing Definition C.1 to Definition 6.9 in two steps. We assume that no two variables in the original expression $e[\vec{\bullet}]_{\mathcal{H}}$ have the same name as this can be enforced by α-renaming.

The first step handles expressions that only substitute values (and have no holes), but allow any number of both secret and untrusted values. An expression of the form in this corollary is easily rewritten as such a substitution as follows. For each hole $[\bullet]_{\mathcal{H}}$, we note that $\Gamma'; pc' \vdash [\bullet]_{\mathcal{H}} : \tau''$ where $\Gamma, \vec{x}:\vec{\tau} \subseteq \Gamma'$ and $pc' \in \mathcal{H}$. We replace the hole with a function application inside a `bind`. Specifically, the hole becomes

$$\texttt{bind}\ y' = y\ \texttt{in}\ \left(y'\ z_1\ \cdots\ z_k\right)$$

where y and y' are fresh variables and the z_is are every variable in $\Gamma' \setminus \Gamma$ (including every element of $\vec{x}$). Let

$$\tau_y = pc'\ \texttt{says}\ \left(\tau_{z_1} \xrightarrow{pc'} \cdots \xrightarrow{pc'} \tau_{z_k} \xrightarrow{pc'} \tau''\right)$$

and include $y:\tau_y$ as the type of an untrusted value to substitute in.

Instead of inserting the expression a into that hole, we substitute in for y the value

$$w = \overline{\eta}_{pc'}\ \left(\lambda(z_1:\tau_{z_1})[pc'].\ \cdots\ \lambda(z_k:\tau_{z_k})[pc'].\ a\right).$$

By Hole we know that $pc' \in \mathcal{H}$ and $\vdash \tau''$ prot $\mathcal{H}$, so the type has the proper protection, and by construction $\Gamma; pc \vdash w : \tau_y$. Moreover, while it has an extra value at the beginning of the trace (the function), the rest of the traces are necessarily the same.

As a second step, we reduce the rest of the way to the expressions used in Definition 6.9. To get from our intermediate step to these single-value expressions, if we wish to substitute k_s secret values and k_u untrusted values, we instead substitute a single list of k_s secret values and a single list of k_u untrusted values. These lists are constructed in the usual way out of pairs, meaning the protection relations continue to hold as required. Finally, whenever a variable is referenced in the unsubstituted expression, we instead select the appropriate element out of the substituted list using nested projections. □

We also note that the same result holds if we allow for insertion of secret code and untrusted values, as the argument is exactly dual. Such a situation, however, makes less sense, so we do not present it explicitly.