Gradual Information Flow Typing

Tim Disney and Cormac Flanagan

University of California Santa Cruz

Abstract. We present a method to support the gradual evolution of secure scripts by formalizing an extension of the simply-typed lambda calculus that provides information flow constructs. These constructs allow initially insecure programs to evolve via targeted refactoring and to provide dynamic information flow guarantees via casts, as well as static information flow guarantees via labeled types.

1 Introduction

Several decades of software engineering experience has demonstrated that writing "correct" software is close to impossible, due to the inherent complexity of software systems and the fallible nature of human programmers. Consequently, relying on security to be an emergent property of software is unwise. We argue instead that security properties, such as data confidentiality and integrity, should be monitored and enforced by small trusted parts of the code base, with help from the run-time system where appropriate.

In practice, programmers are initially concerned more with functionality than with security. It is only once a system has proven useful, and has attracted users and potential attackers, that security concerns become dominant. While it might be preferable to address security concerns right from the start of a project, competitive pressures often dictate quickly developing an initial (perhaps insecure) system that helps clarify the requirements and gain a market foothold, and then to evolve the system with additional features, including security guarantees. Hence, we would like to support a development methodology whereby the programmer first develops an initial, insecure system, and then incrementally refactors the system to add data confidentiality and integrity guarantees via information flow tracking.

Much prior work has addressed information flow security. Most of this work has focused on static type systems such as JFlow [15], Jif [14], and others [29, 16], which involve significant up-front costs. More recent investigations explore dynamic information flow [4, 5, 10], which requires less up-front investment but which cannot document security properties as types.

In this paper, we explore some initial steps towards realizing the vision of gradual evolution from untyped scripts into security typed applications. Since prior work has addressed evolving dynamic scripts into typed code [1, 27, 12, 11], the starting point for our development is a typed language. We explore how to gradually extend programs in this language with security guarantees and with security types.

To support gradual evolution of security properties, our approach provides both static and dynamic information flow guarantees. We use dynamic information flow tracking for all code, including conventionally typed code that has not been refactored to express information flow properties in the type system. Our language includes a *labeling* operation for marking data as private, and a *cast* operation for checking the labels on data. Both operations naturally extend to higher-order data by treating contravariant function arguments appropriately. The cast operation may fail if applied to incorrectly-labeled data; in this case either the term inside the cast or the context of the cast is blamed, which we call positive and negative blame respectively.

For any program, including those without security types, our approach guarantees *termination insensitive non-interference* (TINI), which means that private inputs cannot flow into or influence public outputs. Attempts to violate TINI results in cast failures. We assume a *label lattice* for expressing data confidentiality and integrity properties. For simplicity, we often use a two-element lattice with a public or low confidentiality label (L) and a private or high confidentiality label (H), but the approach generalizes to any lattice.

In addition to tracking information flow dynamically, we enrich the type system to express invariants regarding security labels on the underlying data. Our preservation theorem states that if a term t has type $\texttt{Int}^k$, where k is a security label, then t can only evaluate to a value n^m, where n is an integer and the label m satisfies $m \sqsubseteq k$. Thus, the type system documents a conservative upper bound on the label of the resulting value.

In our language, each value and type has an associated security label. To support legacy code, an unlabeled value implicitly has label $\bot$ (the bottom element in the lattice) indicating that the value is not confidential. Conversely, an unlabeled type implicitly has label $\top$, allowing it to describe values with arbitrary labels, since any label m satisfies $m \sqsubseteq \top$. In this manner, conventionally typed code can interoperate with new precisely typed code, with casts at interfaces between the two typing disciplines.

If the entire application has precise security types, then there is no need for downcasts and for dynamic tracking of information flow labels. This approach has been explored in depth in prior systems such as JFlow [15]. In contrast, the novelty of this paper is that dynamic label tracking enables downcasts, and avoids the need to statically document precise security types throughout the entire application all at once. Instead, the application can gradually evolve from (1) conventionally typed with no security guarantees, to (2) having casts and dynamic information flow guarantees, to (3) being precisely typed with no casts and with static information flow guarantees. Additionally, this evolution process can stop or pause at any point in the middle, depending on engineering, economic, and security requirements, as each intermediate step is a valid well-typed program (albeit with different run-time guarantees).

1.1 Motivating Example

To illustrate the benefits of gradual information flow, consider the following code fragment which deals with sensitive salary information:

$$
\begin{array}{l}
\texttt{let } \mathit{age} : \texttt{Int} = 42 \\
\texttt{let } \mathit{salary} : \texttt{Int} = 58000 \\
\texttt{let } \mathit{intToString} : \texttt{Int} \to \texttt{Str} = \ldots \\
\texttt{let } \mathit{print} : \texttt{Str} \to \texttt{Unit} = \lambda s{:}\texttt{Str}.\ \ldots \\
\mathit{print}(\mathit{intToString}(\mathit{salary}))
\end{array}
$$

This code does not track the flow of sensitive information. After some embarrassing salary leaks, the program manager might want to "harden" the script to limit the flow of sensitive information

In the following revised code, the labeling operation $(58000 : \texttt{Int} \Rrightarrow \texttt{Int}^H)$ marks data as private, and the cast $(s : \texttt{Str} \Rightarrow^p \texttt{Str}^L)$ checks that data is public:

$$
\begin{array}{l}
\texttt{let } \mathit{age} : \texttt{Int} = 42 \\
\texttt{let } \mathit{salary} : \texttt{Int} = 58000 : \texttt{Int} \Rrightarrow \texttt{Int}^H \\
\texttt{let } \mathit{intToString} : \texttt{Int} \to \texttt{Str} = \ldots \\
\texttt{let } \mathit{print} : \texttt{Str} \to \texttt{Unit} = \lambda s{:}\texttt{Str}.\ \texttt{let } s = (s : \texttt{Str} \Rightarrow^p \texttt{Str}^L)\ \texttt{in} \ldots \\
\mathit{print}(\mathit{intToString}(\mathit{salary}))
\end{array}
$$

The runtime system tracks the flow of information through all code. Since the *intToString* library function is applied to a confidential argument, it produces a confidential result, and so the cast inside *print* will fail at runtime. Thus, independent of bugs in the rest of the code, *print* ensures that confidential data is never printed.

As a next step, we wish to document and verify information flow invariants using the type system. We begin by extending the code with explicit labels on types. Note that $\texttt{Int}^H$ is the most general integer type, since these potentially private integers can store both public and private data.

$$
\begin{array}{l}
\texttt{let } \mathit{age} : \texttt{Int}^L = 42 \\
\texttt{let } \mathit{salary} : \texttt{Int}^H = 58000 : \texttt{Int}^L \Rrightarrow \texttt{Int}^H \\
\texttt{let } \mathit{intToString} : \texttt{Int} \to \texttt{Str} = \ldots \qquad \text{// unrefactored module} \\
\texttt{let } \mathit{print} : \texttt{Str}^H \to \texttt{Unit}^H = \lambda s{:}\texttt{Str}^H.\ \texttt{let } s = (s : \texttt{Str}^H \Rightarrow^p \texttt{Str}^L)\ \texttt{in} \ \ldots \\
\mathit{print}(\mathit{intToString}(\mathit{salary}))
\end{array}
$$

The above code incorporates information flow types but does not yet provide static guarantees since *print* accepts H values (at least statically). To provide static guarantees, we first refine *print*'s argument type to specify that it only accepts public data. This refactoring requires introducing a variant of the *intToString* function, called *intToStringL*, for handling public data, using a cast to specify that *intToString* has the desired behavior of mapping public

Figure 1: λ_{gif} Syntax

$\iota ::= \texttt{Int} \mid \texttt{Bool} \mid \texttt{Str}$	*Base Types*
$a, b ::= \iota \mid A \to B$	*Raw Types*
$A, B ::= a^k$	*Labeled Types*
$t, s ::= v \mid x \mid t\ s \mid op\ \overline{t} \mid t\colon A \Rrightarrow B \mid t\colon A \Rightarrow^p B$	*Terms*
$r ::= c \mid \lambda x\colon A.\ t$	*Raw Values*
$v, w ::= r^k$	*Labeled Values*
k, l, m	*Labels*
$\Gamma ::= \emptyset \mid \Gamma, x\colon A$	*Typing Environment*

integer inputs to public string results.

$\texttt{let}\ age : \texttt{Int}^L = 42$
$\texttt{let}\ salary : \texttt{Int}^H = 58000\colon \texttt{Int}^L \Rrightarrow \texttt{Int}^H$
$\texttt{let}\ intToString : \texttt{Int} \to \texttt{Str} = \ldots$ // unrefactored module
$\texttt{let}\ intToStringL : \texttt{Int}^L \to \texttt{Str}^L = intToString\colon (\texttt{Int}^H \to \texttt{Str}^H) \Rightarrow^q (\texttt{Int}^L \to \texttt{Str}^L)$
$\texttt{let}\ print : \texttt{Str}^L \to \texttt{Unit}^L = \lambda s\colon \texttt{Str}^L.\ \ldots$
$print(intToStringL(salary))$

Using these more precise types, bugs such as $print(intToStringL(salary))$ now are revealed at compile time. The programmer then corrects the code to the intended $print(intToStringL(age))$, which passes both static and dynamic security checks. Note that this security typed code interoperates with the legacy *intToString* module via the security interface specification (aka cast) inside *intToStringL*.

2 The Gradual Information Flow Language

We formalize our ideas for gradual security for an idealized language called λ_{gif}, which extends the simply typed λ-calculus with gradual information flow.

The syntax of λ_{gif} is presented in figure 1. Raw types a include integers (`Int`), booleans (`Bool`), strings (`Str`), and function types ($A \to B$). Types A are labeled raw types (a^k). Security labels (k) denote the confidentiality or integrity of a particular value or term. The set of labels form a lattice, with a ordering operation $\sqsubseteq$, join operation $\sqcup$, least element $\bot$, and top element $\top$.

Terms t include variables (x), function applications ($t\ s$), primitive operations ($op\ \overline{t}$), and values (v). Raw values r can be either constants (c), such as 42 or `true`, or functions ($\lambda x\colon A.\ t$). Values v are labeled raw values (r^k). The classification operation ($t\colon A \Rrightarrow B$) adds a label to a value. For example, $3^L : \texttt{Int}^L \Rrightarrow \texttt{Int}^H$ evaluates to 3^H.

Casts ($t\colon A \Rightarrow^p B$) attempt to coerce a term t of type A into a new type B. If the labels on the value are not compatible with type B, the cast will fail, in which case the blame label p assigns blame to the appropriate code fragment. For example, attempting to *downcast* a private integer 42^H to public via the cast $42^H\colon \texttt{Int}^H \Rightarrow^p \texttt{Int}^L$ will fail. An *upcast* of a public integer $42^L\colon \texttt{Int}^L \Rightarrow^p \texttt{Int}^H$ to a private integer however will succeed, and return the value 42^L unchanged. That is, casts do not change values, they only change static types (or else fail).

2.1 Operational Semantics

We formalize the dynamic semantics of λ_{gif} using the big-step evaluation relation $t \Downarrow v$, which evaluates a term t to value v: see figure 2. The [E-APP] rule for function application $(t\ s)$ evaluates t to a function $(\lambda x\colon A.\ t_1)^k$ with a security label k, evaluates the argument s to a value v, and then evaluates the substituted function body $t_1[x := v]$ to a labeled value r^m. The result of the application depends on the function that is invoked, so the rule adds the label k of the callee to the resulting value, yielding $r^{m \sqcup k}$.

The [E-PRIM] rule for primitive operations $(op\ \bar{t})$ refers to the δ_{op} function, which defines the semantics of primitive operations on raw values.

There are three rules to support the *cast* operation, which checks if a runtime label is compatibile with a specified static label. If the check fails then the rules use a *blame label* p to identify the code that is at fault. We say that *positive* blame (p) means the term within the cast is at fault and *negative* blame ($\overline{p}$) means the context containing the cast is at fault. The negation of negative blame is the original blame label: $\overline{\overline{p}} \stackrel{\text{def}}{=} p$.

The [E-CAST-BASE] rule is for casts of base types $\imath$ (i.e. non-functions). The cast $(t\colon \imath^k \Rightarrow^p \imath^l)$ evaluates t to a value r^m and checks that the label m on the value is less than the label l on the target type; if not then the [B-CAST-BAD] rule will blame p. The other [B-...] rules simply propagate blame.

The [E-CAST-FN] rule for $t\colon (A \to B)^k \Rightarrow^p (A' \to B')^l$ is similar to [E-CAST-BASE], except that the value r^m produced by t is wrapped in a new function:

$$(\lambda x'\colon A'.\ (r^m\ (x'\colon A' \Rightarrow^{\overline{p}} A))\colon B \Rightarrow^p B')^{\bot}$$

which satisfies the target type $(A' \to B')$. The wrapper function is used to cast the argument and result to the appropriate types. The argument x' is cast from the new type A' to the original type A, which the original function r can accept. The blame in this cast is inverted $\overline{p}$ to indicate that if this cast fails blame is assigned to the cast context (which invoked the function with an incompatible argument). The result of calling the function is cast to B'.

For an example of the cast rule, consider a function f of type $\texttt{Int}^L \to \texttt{Bool}^L$. If we strengthen its range via the cast $f\colon (\texttt{Int}^L \to \texttt{Bool}^L) \Rightarrow^p (\texttt{Int}^L \to \texttt{Bool}^H)$, calling the resulting wrapper function f' : $\texttt{Int}^L \to \texttt{Int}^H$ will always succeed since the result res of f is guaranteed to be public and f' casts res to a private boolean, which will always succeed. If, however, we strengthen the domain with the cast $f\colon (\texttt{Int}^L \to \texttt{Bool}^L) \Rightarrow^p (\texttt{Int}^H \to \texttt{Bool}^L)$, the argument x' must be downcast $(x'\colon \texttt{Int}^H \Rightarrow^{\overline{p}} \texttt{Int}^L)$ and will fail when x' is private.

The final two rules support *classification*, marking data as having higher confidentiality (or alternatively lower integrity). The [E-CLASSIFY-BASE] rule is used for classifying base types. The classification $t\colon \imath^k \Rrightarrow \imath^l$ adds the target label l to the data by evaluating t to a value r^m and joining l to label m.

The [E-CLASSIFY-FN] rule for $t\colon (A \to B)^k \Rrightarrow (A' \to B')^l$ returns a wrapper function

$$(\lambda x'\colon A'.\ (r^m\ (x'\colon A' \Rrightarrow A))\colon B \Rrightarrow B')^{m \sqcup l}$$

Figure 2: λ_{gif} Operational Semantics

$\boxed{t \Downarrow v}$

$$\frac{}{v \Downarrow v}\ [\text{E-VALUE}] \qquad \frac{t \Downarrow (\lambda x{:}\, A.\ t_1)^k \quad s \Downarrow v \quad t_1[x := v] \Downarrow r^m}{t\ s \Downarrow r^{m \sqcup k}}\ [\text{E-APP}]$$

$$\frac{r = \delta_{op}(r_1, \cdots, r_n) \quad t_i \Downarrow r_i^{k_i} \quad k = \sqcup k_i}{op\ \overline{t} \Downarrow r^k}\ [\text{E-PRIM}] \qquad \frac{t \Downarrow r^m \quad m \sqsubseteq l}{(t{:}\, i^k \Rightarrow^p i^l) \Downarrow r^m}\ [\text{E-CAST-BASE}]$$

$$\frac{t \Downarrow r^m \quad m \sqsubseteq l \quad v = (\lambda x'{:}A'.\ (r^m\ (x'{:}\, A' \Rightarrow^{\overline{p}} A)){:}\, B \Rightarrow^p B')^{\perp}}{(t{:}\, (A \to B)^k \Rightarrow^p (A' \to B')^l) \Downarrow v}\ [\text{E-CAST-FN}]$$

$$\frac{t \Downarrow r^m}{(t{:}\, i^k \Rrightarrow i^l) \Downarrow r^{m \sqcup l}}\ [\text{E-CLASSIFY-BASE}]$$

$$\frac{t \Downarrow r^m \quad v = (\lambda x'{:}A'.\ (r^m\ (x'{:}\, A' \Rrightarrow A)){:}\, B \Rrightarrow B')^{m \sqcup l}}{(t{:}\, (A \to B)^k \Rrightarrow (A' \to B')^l) \Downarrow v}\ [\text{E-CLASSIFY-FN}]$$

$\boxed{t \Downarrow \mathit{blame}\ p}$

$$\frac{t \Downarrow r^m \quad m \not\sqsubseteq l}{(t{:}\, a^k \Rightarrow^p b^l) \Downarrow \mathit{blame}\ p}\ [\text{B-CAST-BAD}] \qquad \frac{t_i \Downarrow v_i \quad \forall i \in 1..j-1 \quad t_j \Downarrow \mathit{blame}\ p}{op\ \overline{t} \Downarrow \mathit{blame}\ p}\ [\text{B-PRIM}]$$

$$\frac{t \Downarrow \mathit{blame}\ p}{t\ s \Downarrow \mathit{blame}\ p}\ [\text{B-APP-L}] \qquad \frac{t \Downarrow v \quad s \Downarrow \mathit{blame}\ p}{t\ s \Downarrow \mathit{blame}\ p}\ [\text{B-APP-R}]$$

$$\frac{t \Downarrow \mathit{blame}\ p}{(t{:}\, a^k \Rightarrow^p b^l) \Downarrow \mathit{blame}\ p}\ [\text{B-CAST}] \qquad \frac{t \Downarrow \mathit{blame}\ p}{(t{:}\, A \Rrightarrow B) \Downarrow \mathit{blame}\ p}\ [\text{B-CLASSIFY}]$$

that adds the labels in A to the argument and the labels in B' to the result. In addition, the security label of the function type is maintained by giving the wrapper function the label from the original function (m) joined with the label from the function being cast to (l).

3 Termination Insensitive Non-Interference

The central guarantee provided by our semantics is non-interference, which informally states that two runs of the same program that differ only in private data will not produce different public results. We formalize the notion of two terms differing only in private data via the equivalence relation ($\sim_H$) defined in figure 3. Essentially, two values are equivalent if either (1) both are at least as secure as H (where H is an arbitrary lattice element) or (2) their subterms are equivalent.

Theorem 1 (Termination Insensitive Non-Interference).
If $t_1 \sim_H t_2$ and $t_1 \Downarrow v_1$ and $t_2 \Downarrow v_2$ then $v_1 \sim_H v_2$.

Figure 3: Equivalence

$\boxed{v \sim_H v}$

$$\frac{H \sqsubseteq m_1 \quad H \sqsubseteq m_2}{r_1^{m_1} \sim_H r_2^{m_2}} \text{ [EQ-VAL1]} \qquad \frac{r_1 \sim_H r_2}{r_1^m \sim_H r_2^m} \text{ [EQ-VAL2]}$$

$\boxed{r \sim_H r}$

$$\frac{t_1 \sim_H t_2}{(\lambda x{:}A.\ t_1) \sim_H (\lambda x{:}A.\ t_2)} \text{ [EQ-FUN]} \qquad \frac{}{c \sim_H c} \text{ [EQ-CONST]}$$

$\boxed{t \sim_H t}$

$$\frac{}{x \sim_H x} \text{ [EQ-VAR]} \qquad \frac{t_1 \sim_H t_2 \quad s_1 \sim_H s_2}{(t_1\ s_1) \sim_H (t_2\ s_2)} \text{ [EQ-APP]}$$

$$\frac{t_1 \sim_H t_2}{(t_1 : A \Rightarrow^p B) \sim_H (t_2 : A \Rightarrow^p B)} \text{ [EQ-CAST]} \qquad \frac{t_i \sim_H t_i' \quad i \in 1..n}{(op\ \overline{t}) \sim_H (op\ \overline{t'})} \text{ [EQ-PRIM]}$$

$$\frac{t_1 \sim_H t_2}{(t_1 : A \Rrightarrow B) \sim_H (t_2 : A \Rrightarrow B)} \text{ [EQ-CLASSIFY]}$$

Proof. By induction on the derivation of $t_1 \Downarrow v_1$ and case analysis on the last rule used in the derivation.

Note that since non-interference is *termination insensitive* two different program runs could differ in their termination behavior (e.g. one could run to normal completion while the other terminates due to an attempted leaking of private data). The termination behavior permits an attacker to learn at most one bit of information about a value per execution[1]. Termination *sensitive* non-interference is a stronger guarantee but requires verifying that every loop with a confidential loop test eventually terminates, which is rather difficult (see for example [7]).

Note that blame is an additional termination channel. We could have two equivalent terms where one term evaluates to a value and the other fails by assigning blame. This does not affect termination insensitive non-interference since assigning blame is just another method of termination.

4 Gradual Information Flow Types

The runtime semantics detects bad downcasts in order to guarantee termination insensitive non-interference. However, we also want to catch security violations at compile time, where possible. To achieve this goal, we next develop a gradual type system where the labels on static types provide an upper bound on the labels of corresponding dynamic values.

[1] Though Askarov et al. [3] point out that an attacker could use intermediary output channels to leak more than a single bit, but only through a brute-force attack

The type system is given by the typing relation $\Gamma \vdash t : A$, which judges a term t to have type A under the typing environment Γ: see figure 4. The [T-PRIM] rule enforces that for each primitive operation $op\ \bar{t}$, the raw types a_i are compatible with the type signature $type(op) : a_1 \times \ldots \times a_n \to b$. It also joins the labels from each argument type ($l = \sqcup l_i$) into the result type b^l so that the resulting type will be at least as secure as the most secure argument.

The [T-APP] rule for function application $(t\ s)$ judges t to have the function type $(A \to b^k)^l$ and the argument s to have a type A' that is a subtype of A. Subtyping allows a function expecting a private input to also accept public arguments, since it will use both safely. In addition, the resulting type b^k is joined with the function's label l since the result depends on the function being used.

The [T-CAST] rule for $t : A \Rightarrow^p B$ enforces that A and B are identical apart from security labels. The operation $\lfloor . \rfloor$ defined in figure 4 strips labels from a λ_{gif} type to consider just the base types. Note that a well-typed cast may fail at runtime if the runtime security labels are not compatible.

The [T-CLASSIFY] rule for $t : A \Rrightarrow B$ checks that B has higher security labels than A. This rule uses the *positive subtyping* relation ($<:^+$) instead of the *standard subtyping* relation ($<:$) since it is not acceptable to lower the security label of a function's domain with a classification. If this rule used standard subtyping, then the classification $t : \texttt{Int}^H \to \texttt{Int}^H \Rrightarrow \texttt{Int}^L \to \texttt{Int}^H$ would be valid, which we do not want.

The full subtyping relation is given in figure 4. Two types are subtypes if they have the same base type and their labels are compatible ($l \sqsubseteq k$). If the types are function types, then the labels must be compatible and the domains must be contravariant ($A' <: A$) and the ranges covariant ($B <: B'$).

The typing system ensures that the labels in each static type is a conservative upper bound on the labels of corresponding runtime values.

We note that if a term t is well typed and we evaluate t then the resulting value v will still be well typed with the same type.[2]

Theorem 2 (Preservation).
If $\emptyset \vdash t : A$ *and* $t \Downarrow v$ *then* $\emptyset \vdash v : A$

The type system defers cast checks to the runtime system, since the safety of downcasts cannot be determined by the typing rules. For example, $v : \texttt{Int}^H \Rightarrow^p \texttt{Int}^L$ will succeed if v has a public runtime label but it will fail if the label is private. However, we can still use types to partially reason about which cast failures may occur. In particular, if two types are subtypes in a cast, then it is not possible for either positive or negative blame to occur. Furthermore, we can decompose the subtyping relation into positive and negative subtyping (see figure 4), in a manner similar to [1, 2, 27]. If the types in a cast are positive (resp. negative) subtypes then the cast cannot produce positive (resp. negative) blame.

[2] Since we are using a big-step semantics to simplify the proof of non-interference we omit the standard progress theorem, which is difficult to show in a big-step semantics.

Figure 4: λ_{gif} Typing Rules

$\boxed{\Gamma \vdash t : A}$

$$\frac{}{\Gamma \vdash c^l : type(c)^l} \text{ [T-CONST]} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \text{ [T-VAR]}$$

$$\frac{\Gamma, x\colon A \vdash t : B}{\Gamma \vdash (\lambda x\colon A.\ t)^l : (A \to B)^l} \text{ [T-ABST]} \qquad \frac{\Gamma \vdash t : A \quad A <:^+ B}{\Gamma \vdash (t\colon A \Rrightarrow B) : B} \text{ [T-CLASSIFY]}$$

$$\frac{\Gamma \vdash t : (A \to b^k)^l \quad \Gamma \vdash s : A' \quad A' <: A}{\Gamma \vdash t\ s : b^{k \sqcup l}} \text{ [T-APP]} \qquad \frac{\Gamma \vdash t : A \quad \lfloor A \rfloor = \lfloor B \rfloor}{\Gamma \vdash (t\colon A \Rightarrow^p B) : B} \text{ [T-CAST]}$$

$$\frac{\Gamma \vdash t_i : a_i^{l_i} \quad i \in 1..n \quad type(op) : a_1 \times \ldots \times a_n \to b \quad l = \sqcup l_i}{\Gamma \vdash op\ \overline{t} : b^l} \text{ [T-PRIM]}$$

$\boxed{A <: B}$

$$\frac{l \sqsubseteq k}{\imath^l <: \imath^k} \text{ [SUB-BASE]} \qquad \frac{l \sqsubseteq k \quad A' <: A \quad B <: B'}{(A \to B)^l <: (A' \to B')^k} \text{ [SUB-APP]}$$

$$\frac{l \sqsubseteq k}{\imath^l <:^+ \imath^k} \text{ [SUB-P-BASE]} \qquad \frac{l \sqsubseteq k \quad A' <:^- A \quad B <:^+ B'}{(A \to B)^l <:^+ (A' \to B')^k} \text{ [SUB-P-APP]}$$

$$\frac{k \sqsubseteq l}{\imath^l <:^- \imath^k} \text{ [SUB-N-BASE]} \qquad \frac{k \sqsubseteq l \quad A' <:^+ A \quad B <:^- B'}{(A \to B)^l <:^- (A' \to B')^k} \text{ [SUB-N-APP]}$$

$\boxed{\lfloor A \rfloor : \lambda_{gif}\ types \to \lambda_{stlc}\ types}$

$$\lfloor (A \to B)^k \rfloor = \lfloor A \rfloor \to \lfloor B \rfloor \qquad \lfloor a^k \rfloor = a$$

Theorem 3 (Blame Theorem).

1. *If* $\emptyset \vdash t : A$ *and* $\forall (t' : B \Rightarrow^p C) \in t$, $B <: C$ *then* $t \not\Downarrow$ *blame* p *and* $t \not\Downarrow$ *blame* $\overline{p}$.
2. *If* $\emptyset \vdash t : A$ *and* $\forall (t' : B \Rightarrow^p C) \in t$, $B <:^+ C$ *then* $t \not\Downarrow$ *blame* p.
3. *If* $\emptyset \vdash t : A$ *and* $\forall (t' : B \Rightarrow^{\overline{p}} C) \in t$, $B <:^- C$ *then* $t \not\Downarrow$ *blame* $\overline{p}$.

Proof. By contradiction assuming that blame has occurred.

5 Related Work

Information flow has a long history of investigating both static and dynamic approaches to track information going back to the work of Denning [8, 9]. Sabelfeld and Myers have an extensive survey of the field [18]. Our paper provides a synthesis of prior static and dynamic techniques.

There are a number of approaches that use type systems for information flow. Volpano et al. [26] formulate the work of Denning as a type system and prove its soundness. Heintze and Riecke [13] extend a simple calculus that uses

a type system to track direct and indirect object creators and readers. Pottier and Simonet [17] present information flow type inference for a simplified ML.

Some approaches use purely dynamic techniques. Austin and Flanagan [4, 5] dynamically track information flow. Shroff et al. [19] dynamically track information flow by tracking indirect dependencies of program points. Devriese and Piessens [10] take an alternative approach called secure multi-execution that runs the program multiple times, once for each security level.

Several approaches use a hybrid of static analysis with dynamic checks during runtime to enforce information-flow guarantees. This idea is similar to our work but our contribution is to allow the programmer to choose when to use dynamic checks and when to use static typing. Chandra and Franz [6] use both static and dynamic techniques for the Java Virtual Machine and allow policies to be changed at runtime. Myers [15] defines an extension to Java called JFlow (which has become Jif [14]) using the hybrid method.

Research on integrating static and dynamic type systems also has a large body of work which we take as our starting point for extending types with security labels. Thatte [22] uses structural subtyping and the notion of *quasi-static* typing to integrate static and dynamic types. Tobin-Hochstadt and Felleisen [23] automatically infer contracts on untyped modules and formulate Typed Racket [24, 25]. Gronski et al. [12] use hybrid type checking, which integrates static type checking with dynamic contract checking in the Sage language. Siek and Taha [20] present gradual typing which uses runtime casts when types are not known at compile time. Wrigstad et al. [28] use the notion of *like types* in the Thorn language. Ahmed et al. [21, 1, 27] combine static and dynamic types with casts and *blame*; much of our formulation follows their methods and notation but with the addition of security labels and information flow.

6 Conclusion

We have presented an idealized language for *gradual security*. The language enables programmers to mark data as confidential, and the language runtime tracks confidential data through all program operations, allowing subsequent cast checks to ensure that sensitive data is not released inappropriately. In this way, termination insensitive non-interference is guaranteed in a dynamic manner.

In addition, types can be gradually refined with security labels to document interface expectations and to statically reason about the data. These labels need not be added all at once; instead, dynamic casts mediate between conventionally typed code (with no security labels) and precisely typed code (with labels).

We show how the notions of positive and negative subtyping help reason about which casts may fail at run-time, and who may be blamed for such failures.

This work represents an initial exploration in terms of an idealized language, illustrating some key ideas and correctness properties. Much work remains to scale up these techniques to realistic languages and to validate the practical utility of gradual security. In particular, we have not yet addressed assignments, which introduce some difficulties for dynamic information flow due to implicit flows, and which remains an important topic for future work.

References

1. A. Ahmed, R. Findler, J. Matthews, and P. Wadler. Blame for all. In *Proceedings for the 1st workshop on Script to Program Evolution*, pages 1–13. ACM, 2009.
2. A. Ahmed, R. Findler, J. G. Siek, and P. Wadler. Blame for all, 2011. Draft copy, to appear in POPL 2011.
3. A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS '08: Proceedings of the 13th European Symposium on Research in Computer Security*, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.
4. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 113–124, New York, NY, USA, 2009. ACM.
5. T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 1–12. ACM, 2010.
6. D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *ACSAC*, pages 463–475. IEEE Computer Society, 2007.
7. B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In *Computer Aided Verification*, pages 328–340. Springer, 2008.
8. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.
9. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
10. D. Devriese and F. Piessens. Noninterference through secure multi-execution. *Security and Privacy, IEEE Symposium on*, 0:109–124, 2010.
11. R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the International Conference on Functional Programming*, pages 48–59, 2002.
12. J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Practical hybrid checking for expressive types and specifications. In *Proceedings of the Workshop on Scheme and Functional Programming*, pages 93–104, 2006.
13. N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Symposium on Principles of Programming Languages*, pages 365–377, 1998.
14. Jif homepage. http://www.cs.cornell.edu/jif/, accessed October 2010.
15. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.
16. A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Symposium on Operating System Principles*, pages 129–142, 1997.
17. F. Pottier and V. Simonet. Information flow inference for ML. *Transactions on Programming Languages and Systems*, 25(1):117–158, 2003.
18. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, Jan 2003.
19. P. Shroff, S. F. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *CSF*, pages 203–217. IEEE Computer Society, 2007.
20. J. G. Siek and W. Taha. Gradual typing for functional languages. In *Proceedings of the Workshop on Scheme and Functional Programming*, 2006.

21. J. G. Siek and P. Wadler. Threesomes, with and without blame. In *POPL*, pages 365–376, 2010.
22. S. Thatte. Quasi-static typing. In *POPL 90 Proceedings of the 17th ACM SIGPLANSIGACT symposium on Principles of programming languages*, pages 367–381. ACM, 1990.
23. S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *Companion to the 21st ACM SIGPLAN symposium on Objectoriented programming systems languages and applications*, pages 964–974. ACM, 2006.
24. S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. *ACM SIGPLAN Notices*, 43(1):395–406, 2008.
25. S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 117–128. ACM, 2010.
26. D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.
27. P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proceedings of the Workshop on Scheme and Functional Programming*, 2007.
28. T. Wrigstad, F. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 377–388. ACM, 2010.
29. S. A. Zdancewic. *Programming languages for information security*. PhD thesis, Cornell University, 2002.