2013 IEEE 26th Computer Security Foundations Symposium

Gradual Security Typing with References

Luminous Fennell, Peter Thiemann
Dept. of Computing Science
University of Freiburg
Freiburg, Germany
Email: {fennell, thiemann}@informatik.uni-freiburg.de

***Abstract*—Type systems for information-flow control (IFC) are often inflexible and too conservative. On the other hand, dynamic run-time monitoring of information flow is flexible and permissive but it is difficult to guarantee robust behavior of a program. Gradual typing for IFC enables the programmer to choose between permissive dynamic checking and a predictable, conservative static type system, where needed.**

We propose ML-GS, a monomorphic ML core language with references and higher-order functions that implements gradual typing for IFC. This language contains security casts, which enable the programmer to transition back and forth between static and dynamic checking.

In particular, ML-GS enables non-trivial casts on reference types so that a reference can be safely used everywhere in a program regardless of whether it was created in a dynamically or statically checked part of the program. The reference can be shared between dynamically and statically checked parts.

We prove the soundness of the gradual security type system along with termination insensitive non-interference.

***Keywords*-gradual typing; security typing; ML; references**

I. Introduction

Language-based analysis and control of information flow in software systems has been studied by numerous authors [13], [23]. Many studies propose type-based, static program analyses where the non-interference property is guaranteed for well-typed programs. Other studies concentrate on dynamic monitoring of program execution where potentially interfering behavior of a program is detected and prevented at run time. Hybrid approaches (e.g., [12], [22]) combine both kinds of analysis.

Static approaches are advantageous for the specification of security policies that are known up-front and where the program can be built to suit the analysis. The analysis guarantees that no security mismatches happen at run time. However, security policies are often formulated as an afterthought, when a large part of a system is already implemented, or they may evolve as the implications of a system design become understood. Unfortunately, a program that was not built with the static analysis in mind can be difficult to modify so that it accepted by the analysis even though the program respects the desired security policies.

Dynamic approaches, on the other hand, enforce a safe approximation to the non-interference property. They do not give static guarantees, but are amenable to changes in the security policy without requiring a rewrite of the code.

The typing community studies a similar interplay between static and dynamic checking. Gradual typing [25], [26] is an approach where explicit cast operations mediate between coexisting statically and dynamically typed program fragments. The type of a cast operation reflects the outcome of the run-time test performed by the cast: The return type of the cast is the compile-time manifestation of the positive test; a negative outcome of the test causes a run-time exception.

Disney and Flanagan propose an integration of security types with dynamic monitoring via gradual typing [11]. Such an integration enables the stepwise introduction of checked security policies into a software system. The programmer inserts checks for these policies as run-time casts at strategic points in the code. The type system statically guarantees adherence to the policies "on the static side" of a cast, whereas the run-time system checks the policies "on the dynamic side".

This procedure creates statically checked regions in a dynamically checked environment. These regions can be enlarged by rewriting code and moving casts to make programs more efficient and to avoid potential run-time errors.

Alternatively, a programmer may impose a static security typing discipline on a program and revert to dynamic checking by inserting casts demarcating the regions where the static checker fails. This approach leads to dynamically checked regions and the programmer should strive to restrict them to places where the static analysis would be too conservative or where the code contains language features not supported by the analysis. The programmer may also restrict debugging efforts that chase policy violations to the dynamically checked parts.

A. Contributions

Disney and Flanagan [11] consider a pure lambda calculus. Building on their ideas, we study a monomorphic ML core language with references. The consideration of references introduces significant complications, as references enable flow-sensitivity attacks [22]. The underlying type system is inspired by Pottier and Simonet's work on a security type system for CoreML [21]. We extend this static system with casts and suitable run-time representations of security levels on the dynamic side of a cast. Our casts are very powerful because they are able to change the security

DOI 10.1109/CSF.2013.22

type of the *content* of a reference. This choice enables our calculus to safely perform casts between security types that are not related by the subtype relation induced by the ordering on security levels: A sound subtype relation must treat references invariantly. The extended calculus uses a no-sensitive-upgrade strategy (NSU, [3]) for information flow control (IFC) on the dynamic side and we prove termination insensitive non-interference for the combined system.

B. Terminology

Security levels are drawn from a two-element lattice with elements L for low-security, public information and H for high-security, secret information. They are ordered by $\mathsf{L} \sqsubset \mathsf{H}$ and the operator $\sqcup$ denotes the least upper bound. Our results generalize to arbitrary discrete security lattices as outlined for CoreML [21]. In the remainder of the paper, we will use a capital "B" as a meta-variable for the security levels of values and a capital "PC" for the *security context*. A security context is the security level of the program counter at run-time.

We write $\texttt{int}^{\mathsf{L}}$ for the type of low-security integers and $\texttt{ref}^{\mathsf{L}}\ \texttt{int}^{\mathsf{H}}$ for the type of low-security pointers that point to high-security integers in memory. The type $(\texttt{int}^{\mathsf{H}} \xrightarrow{\mathsf{L}} \texttt{int}^{\mathsf{H}})^{\mathsf{L}}$ specifies a low-security function that takes a high-security argument and yields a high-security result. The annotation on the arrow restricts the *effect* of the function, i.e., the function allocating or modifying a memory cell, to cells of at least the level indicated by the annotation. The annotation L means that all cells may be modified, whereas H would restrict modification to high-security memory cells.

Upgrading a memory cell means to overwrite its low-security content with a high-security value. Because upgrades that occur in high-security contexts may leak confidential information through implicit flows, they have to be treated specially by dynamic IFC techniques [5]. Security type systems typically forbid upgrades altogether.

We will employ the *non-sensitive-upgrade* (NSU) strategy of Austin and Flanagan [3] as a dynamic IFC technique. In an NSU semantics, assignments fail if they would upgrade a memory cell under a high-security context.

II. Motivation

The following examples illustrate the creation of dynamically checked regions in otherwise statically checked programs and vice versa. The motivation for the examples is to work around restrictions imposed by the underlying security type system, for example the dynamic, manual administering of access rights where the code checks a run-time representation of the accessor's security level.

To start off, consider a report processing application that is statically verified against a standard security type system like that of FlowCaml [21]. It contains numerous security-typed procedures that process reports by reference without requiring any run-time checks. Here are two examples of such procedures with their type signatures:[1]

```
sendToManager : ref Report^H -H-> unit
sendToFacebook : ref Report^L -L-> unit
```

The `sendToManager` function takes a confidential report and guarantees that it is not leaked to the public. The H on the arrow indicates the lowest security level that is modified. The `sendToFacebook` function takes a public report and publishes it to an open Internet message board, as indicated by the low-security effect $\xrightarrow{\mathsf{L}}$.

```
1 (* Some privileged information *)
2 let infoH : ref Report^H = ...
3
4 (* Optionally enhance a report
5  * with privileged information *)
6 addPrivileged isPrivileged worker report =
7   if isPrivileged
8      then report := report + !infoH;
9   worker report
```

Listing 1. Example function with manual security enforcement.

Listing 1 illustrates a proposed extension of the report processing application. It contains a utility function `addPrivileged` that adds privileged information to a report before passing it to a procedure like `sendToManager` or `sendToFacebook`. The flag `isPrivileged` indicates if the `worker` argument is sufficiently trusted to handle a privileged report. If `isPrivileged` is true, then the privileged information is retrieved through a global reference `infoH` and added to `report`. Otherwise, `worker` is called with an unmodified report.

Listing 1 type checks in FlowCaml and also the term

```
addPrivileged true sendToManager
```

is accepted with type $\texttt{ref Report}^{\mathsf{H}} \xrightarrow{\mathsf{H}} \texttt{unit}$, as `sendToManager` is safe for consuming high-security information. In contrast, the term

```
addPrivileged false sendToFacebook
```

is rejected by the type checker, even though it is semantically safe. No privileged information is leaked, but the type checker does not track the correspondence between the security level of the worker and the `isPrivileged` flag.

Our proposal for gradual security typing allows us to embed a (security-) untyped code fragment in a typed setting. A cast expression asserts the required security type signature for the untyped code and the dynamic semantics guarantees adherence to the type by checking for violations of the type signature using dynamic IFC techniques. In our

[1]Type signatures are simplified by omitting some L annotations. The annotation on the unit type does not matter because it cannot convey information in our setting.

type language, the annotation "?" stands for a single security level, unknown from the static point of view, for objects that will be dynamically checked. With this understanding, the utility function addPrivileged can be given the type $\texttt{bool}^{?} \xrightarrow{?} (\texttt{ref Report}^{?} \xrightarrow{?} \texttt{unit}) \xrightarrow{?} \texttt{ref Report}^{?} \xrightarrow{?} \texttt{unit}$ where all security levels (except the implicit L on the references) are checked at run time using dynamic IFC techniques, as advertised by the "?". A programmer must insert casts to use the function addPrivileged with sendToManager and sendToFacebook as shown in the following safe code fragment.[2]

```
let w = {ref Report? →? unit}
          sendToFacebook in
{ref ReportL →L unit}(addPrivileged false w)
```

The first cast $\{\texttt{ref Report}^{?} \xrightarrow{?} \texttt{unit}\}$ wraps the statically checked sendToFacebook function such that the resulting function w expects a report with a run-time tag that indicates its security level. The run-time system also passes a run-time value that indicates the lower bound on the effect of w. The cast operation checks that the run-time tag on the report is L and strips it off, conceptually speaking. It further tests that the allowed effect is L, as expected by sendToFacebook.

The cast $\{\texttt{ref Report}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \texttt{unit}\}$ in this example applies to a function type and is thus delayed to the point where the function is applied. Then the argument is cast according to $\texttt{ref Report}^{\mathsf{L}}$ and the context is also cast to L so that all effects are allowed. This cast reifies static information by attaching or modifying run-time tags on values.[3]

In contrast, the following code fragment shows an example, where a mismatch of the isPrivileged flag with the actual security labels causes a run-time exception.

```
{ref ReportL →L unit}(addPrivileged true w)
```

In many cases, the code can be transformed to make it amenable to a FlowCaml-style type checker or to use simpler casts (i.e., no casts on reference types). To make the addPrivileged example work, two specialized copies of addPrivileged are needed, one where isPrivileged is true and another where it is false. It is one of our points that the use of gradual security typing often avoids such code duplication, thus improving software quality.

As another example, consider the program fragment in Listing 2, which declares two statically type checked procedures in lines 5 and 6. Suppose that they must share a single buffer buf because of resource constraints. A programmer who has to avoid information leaks between wL and wH interposes the function wrap to clear the buffer before passing it to a procedure. While the function wrap is not

[2]In a cast $\{t\}e$, the t is the target type of the cast and e is a term. In the formal system (Sec. III), a cast also mentions the source type.

[3]The actual situation is more complicated because of our liberal handling of reference casts.

```
1  wrap f buf =
2    buf := 0L; f buf
3
4  let buf : ref BufL = newL 0L in
5  let wL  : ref BufL →L unit = ... in
6  let wH  : ref BufH →H unit = ... in
7  ...
8  let wLd = {ref Buf? →? unit} wL in
9  let wHd = {ref Buf? →? unit} wH in
10 ...
11 wrap wLd buf
12 wrap wHd buf
13 ...
```

Listing 2. Example function with manual security enforcement.

acceptable in a type system like that of FlowCaml, our gradual system accepts it with a suitable dynamic type.

Using the definitions of Listing 2 it is easily possible to provoke a security exception:

```
let h : boolH = ... in
...
if h then wrap wLd buf
```

As the execution of wLd depends on the high-security value h, all low-security effects executed by wLd are potential security violations. Our dynamic semantics prevents such security leaks by aborting execution and issuing a security exception.

Although the error that we just provoked occurs during the execution of the typed body of wL, a gradually typed system has significant advantages over a purely dynamic one during development. Suppose that the unsafe code above is the result of a programming error that occurred during the integration of the shared buffer in the system. As wL and wH are fully typed, the developers can restrict their debugging efforts to the dynamic fragment of the system.

The examples up to now demonstrate uses of statically typed procedures in a dynamic context. The converse is also possible. Consider a procedure that formats a document (Listing 3). This procedure should work alike on high and low security documents without leaking. For that reason it is defined on a dynamic document type. Using appropriate casts, the dynamic formatter can be applied without affecting the static security guarantees for the documents. Both casts arising in lines 6 and 7 are reference casts that modify the pointer to dereference to a document with dynamic security level. If the formatting depends on further parameters of unclear security status, then these parameters could be made dynamic in the same way, thus leaving the detection of problems to testing.

```
let format : ref Doc^? --?--> unit = ...

let docL : ref Doc^L = ...
let docH : ref Doc^H = ...

format ({ref Doc^?} docL);
format ({ref Doc^?} docH);
```

Listing 3. Formatting low and high security documents.

$$
\begin{array}{rrl}
\textit{Security Levels} & B, PC & ::= \mathsf{L} \mid \mathsf{H} \\
\textit{Type Annot.} & b, pc & ::= B \mid ? \\
\textit{Raw Types} & s & ::= \texttt{int} \mid \texttt{unit} \mid t \xrightarrow{pc} t \mid \texttt{ref}\ \ t \\
\textit{Types} & t & ::= s^b \\
\textit{Blame Id} & P & ::= \text{(unique identifiers)} \\
\textit{Blame Labels} & p, q & ::= P \mid p, P \\
\textit{Variables} & x & ::= \text{(unique identifiers)} \\
\textit{Constants} & k & ::= 0 \mid 1 \mid 2 \mid \ldots \\
\textit{Raw Values} & w & ::= \lambda x.\, e \mid k \mid () \mid l^{(t,p)} \\
\textit{Values} & v & ::= w^B \\
\textit{Expressions} & e & ::= v \mid x \mid e\, e \\
 & & \mid \texttt{new}^{t,B} e \mid !\ e \mid e := e \\
 & & \mid \{t \Leftarrow t\}^p e \mid \texttt{prot}^B e \\
 & & \mid \{[]b \Leftarrow []b\}^p e \\
 & & \mid \{\texttt{ref}\ \ pc \Leftarrow pc\}^p e \mid \{* \overset{pc \Leftarrow pc}{\rightarrow} *\}^p e
\end{array}
$$

Figure 1. Syntactic domains of ML-GS.

III. An ML Core Language with Gradual Security

Fig. 1 defines the syntax of ML-GS, an ML core language with gradual security. Types are part of the expression syntax because they appear in casts. A type t consists of a raw type s and a type annotation b, which approximates a run-time security level. A type annotation is either a security level, B, or *dynamic*, $?$. The $?$ is a static security level that represents objects that will be dynamically checked. It is treated as a new top element in static checking such that $\mathsf{H} \sqsubset ?$.

A raw type is either the integer type `int`, a function type $t \xrightarrow{pc} t'$, the unit type `unit`, or a reference type `ref` t. The *program counter security level*, pc, on a function type indicates the minimum permitted security level for the function's effects. The significance of pc for typing and execution of ML-GS programs is explained in Sec. IV and Sec. V.

A value is a raw value labeled with its run-time security level B, which starts off as the level of its originating principal and which can be checked during execution. Raw values are lambda abstractions $\lambda x.\, e$, integer constants k, unit values $()$, and pointers $l^{(t,p)}$. Pointers do not appear in source programs and are explained at the end of this section. For readability, we write $\lambda^B x.\, e$ instead of $(\lambda x.\, e)^B$. Let expressions and sequencing are defined as syntactic sugar: `let` $x = e_1$ `in` e_2 desugars to $(\lambda^{\mathsf{L}} x.\, e_2)\, e_1$ and $(e_1; e_2)$ to `let` $z = e_1$ `in` e_2 where z is a fresh variable.

Further expressions are function application $e_1\, e_2$, allocation and initialization of a new heap cell $\texttt{new}^{t,B} e$ (where B is the security level of the pointer and the type t represents the initial type of the cell's content), dereference of an pointer $!\ e$, and assignment of a new value to a cell $e := e$.

The expression $\{t' \Leftarrow t\}^p e$ casts the type of e from source type t to target type t'. Only the target type is needed for the run-time safety checks, but we include the source type to distinguish safe casts from unsafe ones (Sec. VI). A cast carries a *blame label*, a non-empty set p of blame ids P. As in work on contracts [1], [29], a blame id corresponds to the location of a cast in the source code. If the cast leads to a run-time failure at a boundary between static and dynamic checking, then its blame id (i.e., source location) is part of the blame label reported by the failure. Initially, each cast carries its source location as a singleton blame label P. During execution casts with joined blame labels arise. We write pq for the join of two blame labels p and q.

The remaining expressions are all introduced by reductions and do not occur in source programs. The protect expression $\texttt{prot}^B e$ ensures dynamically that the security level of the result of e is at least B and that no effect occurs at a level less than B. Protect expressions from the literature often only do not place a restriction on the effect [14].

The *guard casts* $\{[]pc \Leftarrow []pc'\}^p e$, *function guard casts* $\{* \overset{pc \Leftarrow pc'}{\rightarrow} *\}^p e$, and *pointer casts* $\{\texttt{ref}\ \ pc \Leftarrow pc'\}^p$ are technical devices that ensure type safety during the reduction of casts. They are explained along with their typing and reduction rules in Sec. IV.

The raw pointer value $l^{(t,p)}$ consists of an address l, an *access type* t, and a blame label p. The access type t is used for typing the dereference operation. Its annotations may be different from the *current type* of the value stored at address l. A reference cast changes the access type of a pointer without affecting the current type and a subsequent assignment through this pointer synchronizes the current type with the access type. The *blame label* p tracks the blame ids of such casts. A dereference operation on a pointer will only succeed if its access type is at least as secure as the current type of the referenced memory cell. As a short-hand, we write $l^{(t,q),B}$ for ${l^{(t,p)}}^B$.

IV. Gradual Security Typing

Most work on gradual typing is geared towards dynamic languages where casts serve to discover the concealed structure of run-time values and manifest it in the types. In our work, the erasure of security levels in types results in a simply typed program. Only the run-time security

Variable Env. $\Gamma ::= \text{-} \mid \Gamma, x : t$

Address Env. $M ::= \text{-} \mid M, l : t$

$\boxed{pc; \Gamma; M \vdash e : t}$

T-Var
$$pc; \Gamma, x : t; M \vdash x : t$$

T-Int
$$pc; \Gamma; M \vdash k^B : \mathtt{int}^B$$

T-Unit
$$pc; \Gamma; M \vdash ()^B : \mathtt{unit}^B$$

T-Addr
$$\frac{t \sim t'}{pc; \Gamma; M, l : t \vdash l^{(t',p),B} : \mathtt{ref}^{\,B}\ t'}$$

T-Protect
$$\frac{pc \sqcup B; \Gamma; M \vdash e : s^b}{pc; \Gamma; M \vdash \mathtt{prot}^B e : s^{b \sqcup B}}$$

T-Abs
$$\frac{pc'; \Gamma, x : t; M \vdash e : t'}{pc; \Gamma; M \vdash \lambda^B x.\, e : (t \xrightarrow{pc'} t')^B}$$

T-App
$$\frac{pc; \Gamma; M \vdash e_1 : (t \xrightarrow{pc \sqcup b} s^{b'})^b \qquad pc; \Gamma; M \vdash e_2 : t}{pc; \Gamma; M \vdash e_1\ e_2 : s^{b' \sqcup b}}$$

T-New
$$\frac{pc; \Gamma; M \vdash e : s^b \qquad pc \sqsubseteq b}{pc; \Gamma; M \vdash \mathtt{new}^{s^b, B} e : \mathtt{ref}^{\,B}\ s^b}$$

T-Deref
$$\frac{pc; \Gamma; M \vdash e : \mathtt{ref}^{\,b}\ s^{b'}}{pc; \Gamma; M \vdash !\, e : s^{b' \sqcup b}}$$

T-Asgn
$$\frac{pc \sqcup b \sqsubseteq b' \qquad pc; \Gamma; M \vdash e_1 : \mathtt{ref}^{\,b}\ s^{b'} \qquad pc; \Gamma; M \vdash e_2 : s^{b'}}{pc; \Gamma; M \vdash e_1 := e_2 : \mathtt{unit}^{b'}}$$

T-Sub
$$\frac{t \prec t' \qquad pc' \sqsubseteq pc \qquad pc; \Gamma; M \vdash e : t}{pc'; \Gamma; M \vdash e : t'}$$

T-Cast
$$\frac{pc; \Gamma; M \vdash e : t \qquad t \sim t'}{pc; \Gamma; M \vdash \{t' \Leftarrow t\}^p e : t'}$$

Figure 2. Typing rules.

properties of the values may be concealed in the types by using the dynamic annotation ?. The system also supports standard security subtyping [14], [21] which allows low-security information to be implicitly promoted to a high security level and functions with high-security pc to be

$\boxed{b \prec b'}$ $\boxed{t \prec t'}$

$$\frac{b \sqsubseteq b'}{\mathtt{int}^b \prec \mathtt{int}^{b'}} \qquad \frac{b \sqsubseteq b'}{\mathtt{unit}^b \prec \mathtt{unit}^{b'}} \qquad \frac{b \sqsubseteq b'}{\mathtt{ref}^{\,b}\ t \prec \mathtt{ref}^{\,b'}\ t}$$

$$\frac{b \sqsubseteq b' \qquad pc' \sqsubseteq pc \qquad t_1' \prec t_1 \qquad t_2 \prec t_2'}{(t_1 \xrightarrow{pc} t_2)^b \prec (t_1' \xrightarrow{pc'} t_2')^{b'}}$$

$\boxed{t \sim t'}$

$$\frac{t_1' \sim t_1 \qquad t_2 \sim t_2'}{(t_1 \xrightarrow{pc} t_2)^b \sim (t_1' \xrightarrow{pc'} t_2')^{b'}} \qquad \mathtt{int}^b \sim \mathtt{int}^{b'}$$

$$\mathtt{unit}^b \sim \mathtt{unit}^{b'} \qquad \frac{t \sim t'}{\mathtt{ref}^{\,b}\ t \sim \mathtt{ref}^{\,b'}\ t'}$$

Figure 3. Subtyping and compatibility.

implicitly weakened to functions with low-security pc. The dynamic annotation becomes the most general security level; it subsumes statically low- and high-security annotations.

A type environment Γ associates variables with types whereas an address environment M provides types for heap addresses (see Fig. 2). Fig. 2 also defines the rules for the typing judgment $pc; \Gamma; M \vdash e : t$ for all source expressions (Fig. 8 contains rules for some special expressions that occur only at run time). It relates a pc, a type environment, an address environment and an expression with a type. The program counter security level pc restricts the write effects of a typed expression e: The execution of e may only modify memory that is at least as secure as pc.

The rules are based on the principles of secure information flow [14], [21]: the security level of the result of a computation must be greater than the levels of the arguments and the security level of information that escapes via side effects must not be less than the pc.

The typing rule for variables, *T-Var*, is standard. By rule *T-Int*, an integer constant has the underlying type `int` and a type annotation matching the value. Similarly, the unit value has a type annotation matching its security level. The rule *T-Addr* types a pointer $l^{(t',p),B}$ according to its security level B and its access type t'. Furthermore, the address environment associates the address l with a type t that is compatible to t'. The compatibility requirement $t \sim t'$ forces t and t' to have the same structure, but with potentially different annotations (see Fig. 3).

The protect expression $\mathtt{prot}^B e$ ensures a minimum security level for its argument $e : s^b$ by joining its type's level b with B. In particular, if any of the annotations is dynamic, the result is also dynamic. The *T-Protect* rule also enforces a lower bound on the pc of the protected expression e.

The unannotated part of rule *T-Abs* is standard. The

function type annotation is derived from the security level, B, of the abstraction. Furthermore, the arrow carries a program counter security level, pc', indicating the maximum pc under which the function can be called. As abstractions are values, the program counter security level pc, which the abstraction is typed under, is irrelevant.

The rule for function application *T-App* checks for a function type and a matching type for the argument as usual. The join of the function's security annotation, b, and that of its result type, b', is sufficient to protect the application's result. The function's pc has to respect the join of b and the current pc.

The rules for side-effecting expressions are mostly standard. The security level of the pointer in which information is written is protected by the pc (rules *T-New* and *T-Asgn*). Reading from memory requires the result to subsume the security level of the pointer (rule *T-Deref*). The rule *T-Asgn* requires that the updated memory content is more secure than the current pc and the pointer's annotation.

Rule *T-Sub* enables standard security subtyping. The subtyping relation $\_ \prec \_$ (see Fig. 3) lifts the ordering on type annotations to types, with contravariant function parameter types and invariant reference content types, as usual. The program counter security level on functions is contravariant with respect to the security lattice: If an expression can execute under a high-security pc, it can also execute in an unrestricted environment.

The rule *T-Cast* converts a value of type t to a value of type t' if these types are compatible. Compatibility is an equivalence relation that identifies types solely by their structure, ignoring security annotations; any two types that only differ in their security annotations can be cast into each other. In particular, compatibility subsumes the subtyping relation. For example, $\texttt{int}^{\mathsf{L}} \sim \texttt{int}^{\mathsf{H}}$ and $\texttt{int}^{\mathsf{H}} \sim \texttt{int}^{\mathsf{L}}$ and therefore $\{\texttt{int}^{\mathsf{L}} \Leftarrow \texttt{int}^{\mathsf{H}}\}^p e$, $\{\texttt{int}^{\mathsf{H}} \Leftarrow \texttt{int}^{\mathsf{L}}\}^p e$ are both valid casts (although the former will raise an error at runtime). The subtyping relation only holds for $\texttt{int}^{\mathsf{L}} \prec \texttt{int}^{\mathsf{H}}$. In contrast, the cast $\{\texttt{int}^{\mathsf{H}} \Leftarrow \texttt{ref}^{\mathsf{L}}\ \texttt{int}^{\mathsf{H}}\}^p e$ is invalid because references are always incompatible with integers.

The typing rules for guard casts and pointer casts are discussed in Sec. V in the context of cast reduction, where these expressions arise.

V. Semantics

Before delving into the details of the formal semantics, we give an overview of the execution model. The operational semantics keeps track of security levels exactly as indicated in the syntax: each value carries its security level at run time and non-interference is checked according to the NSU policy all the time, regardless of the guarantees given by the type system.

For the fragment of the calculus that disallows casts on reference types, it would be possible to define an erasure semantics that executes the statically typed parts without security level annotations at run time. In this fragment, casts would erase or add these annotations as appropriate.

$$
\begin{array}{ll}
\textit{Ev. Ctx} & \mathscr{E} ::= [\,]\ e \mid v\ [\,] \mid \{t \Leftarrow t\}^p[\,] \\
& \quad \mid \texttt{new}^{t,B}[\,] \mid [\,] := e \mid v := [\,] \mid\ !\,[\,] \\
& \quad \mid \{* \overset{pc \Leftarrow pc}{\rightarrow} *\}^p[\,] \mid \{\texttt{ref}\ \ pc \Leftarrow pc\}^p[\,] \\
\textit{Heaps} & \mu ::= \text{-} \mid \mu, (l \mapsto \{t\}^p v) \\
\textit{Results} & r ::= e \,/\, \mu \mid \Uparrow p
\end{array}
$$

Figure 4. Semantic domains of ML-GS.

However, reference casts that modify the type of the stored value complicate such an erasure transformation because a number of mutually compatible types may be associated with a single memory address. Directly after allocation, there is one pointer to a fresh address, and the current type and the access type of this address coincide. Next, a cast may change the access type without updating the value so that a subsequent dereference operation expects a differently typed value. At this point, run-time information is needed to check that the value is acceptable at the new access type. The operational semantics inserts a cast at this point.

If we consider this scenario in the context of a hypothetical erasure semantics, the problem comes up immediately. In particular, the initial allocation may happen in a fully statically checked setting so that the stored value carries no annotations. Then the reference content is cast to dynamic. (In our semantics, this cast only affects the type of the reference, but not the stored value. After all, there might be a static alias of the pointer.) The subsequent dereference operation needs to produce a fully annotated value, but they are not available due to erasure.

The presence of references also makes it hard to give a clear boundary between statically and dynamically checked code. In principle, run-time errors due to security mismatches may happen at downcasts, e.g. from ? to L, and at assignments that fail the NSU check. While the assignments may be flagged according to the dynamic annotation of their pointer, the downcasts may not be present in the program from the start. As just discussed, they may arise from dereference operations where the semantics also inserts casts. Thus, any dereference operation that may dereference a cast pointer needs to be checked dynamically.

Thus, as a first approximation, we adopt a fully annotated execution model, knowing that the outcome of a significant fraction of the run-time checks is statically determined by the type system.

A. Configurations

The operational semantics of ML-GS is defined by a reduction relation $e \,/\, \mu \xrightarrow{PC} r$ between configurations $e \,/\, \mu$, which are pairs of an expression and a heap, and results r, which are either configurations or blame exceptions, indexed

$\boxed{e \,/\, \mu \xrightarrow{PC} r}$

R-Ctxt
$$\frac{e \,/\, \mu \xrightarrow{PC} e' \,/\, \mu'}{\mathscr{E}[e] \,/\, \mu \xrightarrow{PC} \mathscr{E}[e'] \,/\, \mu'}$$

R-App
$$(\lambda^B x.\, e)\ v \,/\, \mu \xrightarrow{PC} \mathtt{prot}^B(e[x \mapsto v]) \,/\, \mu$$

R-Protect
$$\mathtt{prot}^B w^{B'} \,/\, \mu \xrightarrow{PC} w^{B' \sqcup B} \,/\, \mu$$

R-Ctxt-Protect
$$\frac{e \,/\, \mu \xrightarrow{PC \sqcup B} e' \,/\, \mu'}{\mathtt{prot}^B e \,/\, \mu \xrightarrow{PC} \mathtt{prot}^B e' \,/\, \mu'}$$

R-New
$$\frac{l \notin \mathrm{dom}(\mu) \qquad p \text{ fresh} \qquad \mu' = \mu, (l \mapsto \{t\}^p w^{B' \sqcup PC})}{\mathtt{new}^{t,B} w^{B'} \,/\, \mu \xrightarrow{PC} l^{(t,p),B} \,/\, \mu'}$$

R-Deref
$$\frac{\mu = \mu', (l \mapsto \{t_2\}^p v)}{!\, l^{(t_2',q),B} \,/\, \mu \xrightarrow{PC} \mathtt{prot}^B \{t_2' \Leftarrow t_2\}^{pq} v \,/\, \mu}$$

R-Asgn
$$\frac{PC \sqcup B \sqsubseteq B_1 \qquad \mu = \mu'', (l \mapsto \{s_1^{b_1}\}^p w_1^{B_1}) \qquad \mu' = \mu'', (l \mapsto \{s^{b_2 \sqcup PC \sqcup B}\}^{pq} w^{B_2 \sqcup PC \sqcup B})}{l^{(s^{b_2},q),B} := w^{B_2} \,/\, \mu \xrightarrow{PC} ()^{PC} \,/\, \mu'}$$

Figure 5. Semantics.

by a PC, which indicates the current security context. It is the dynamic counterpart to the program counter security level pc in the typing rules. The heap μ maps an address l to a combination of a value, a blame label, and a type, written $\{t\}^p v$. Here, t is the *current type* of the value v stored in the cell. The blame label p comes into play when the cell is dereferenced with an access type which is not a subtype of the current type. This subtype relation is checked with a cast labeled with p.

A blame exception $\Uparrow p$ flags the violation of a typing assumption. The blame label p contains the blame id of the responsible cast.

B. Lambda calculus fragment

Figure 4 formally defines results, heaps, and evaluation contexts $\mathscr{E}$ that guide the search for redexes in a standard call-by-value evaluation step.

Figures 5, 6, and 9 contain the reduction rules of ML-GS. The rules in Fig. 5 cover non-cast expressions. The context rule *R-Ctxt* is standard. The application of a lambda abstraction to a value (rule *R-App*) is reduced to the protected function body with the argument value substituted for the formal parameter, using standard, capture-free substitution. The resulting $\mathtt{prot}^B$ expression reduces according to *R-Protect* as soon as its argument becomes a value. If $\mathtt{prot}^B$ appears in the context of a reduction, rule *R-Ctxt-Protect* raises the pc of the execution.

In the interplay of the rules *R-Protect* and *R-App*, the context $\mathtt{prot}^B$ reflects the security constraint of the *T-App* typing rule for the result of the application ($s^{b \sqcup b'}$, cf. Fig. 2). In a typed, cast-free program, the $\mathtt{prot}^B$ context could be omitted because the typing rules ensure the necessary protection of the result. In ML-GS, $\mathtt{prot}^B$ is needed in a dynamic subcomputation which is not restricted by typing.

C. References

The rules *R-New*, *R-Deref*, and *R-Asgn* allocate a new memory cell, dereference a pointer, and assign to a pointer. Additionally they pursue two further objectives: On the one hand they implement dynamic information flow control by checking the value annotations against the current dynamic pc. On the other hand, they manipulate the current types of heap values to discover inconsistencies that may be introduced by reference casts. In the following, we discuss both aspects in turn.

The constraints on security levels in the reduction rules reflect the constraints on type annotations enforced by the respective typing rules. Allocating a value protects it with the current pc. A value that is read from the heap needs to be protected with the pointer's security level.

A heap update (rule *R-Asgn*) has to pass the NSU check: the security level of the value on the heap that is updated, B_2, has to subsume that of the security context, PC, and that of the pointer, B. The updated heap stores the raw value w with a security level that is sufficiently high to respect the context PC and B. The cast failure rules, which are described in Sec. V-D, cover the case when the update value is more secure than the original one and the NSU check fails.

Next we turn to the tracking and checking of reference casts. For a newly allocated reference, the access type and the current type coincide. At this point, the initial blame label p does not matter because it is not associated with a type change. A cast may change the access type (leaving its blame label at the modified pointer) and an assignment may change the current type. In rule *R-Deref*, the access type, t_2', of the pointer, which is used in typing the pointer, may differ from the current type t_2 of the stored value v. Thus, v is retrieved from the heap and cast from its current type t_2 to the expected type t_2'. The blame labels on access and current type are joined on this cast, because both a preceding assignment or a preceding cast may have caused the mismatch between the access type and the current type. The *R-Asgn* rule allows updates that change the type annotations of the stored value. It implements NSU and refuses to overwrite a

$\boxed{e \,/\, \mu \xrightarrow{PC} r}$ cont.

R-Ctxt-Fail
$$\frac{e \,/\, \mu \xrightarrow{PC} \Uparrow p}{\mathscr{E}[e] \,/\, \mu \xrightarrow{PC} \Uparrow p}$$

R-Ctxt-Protect-Fail
$$\frac{e \,/\, \mu \xrightarrow{PC \sqcup B} \Uparrow p}{\mathtt{prot}^B e \,/\, \mu \xrightarrow{PC} \Uparrow p}$$

R-Cast-Sub-Fail
$$\frac{B \not\sqsubseteq b_1}{\{s_1^{b_1} \Leftarrow s_2^{b_2}\}^p w^B \,/\, \mu \xrightarrow{PC} \Uparrow p}$$

R-Asgn-NSU-Fail
$$\frac{PC \sqcup B \not\sqsubseteq B' \qquad \mu = \mu', (l \mapsto \{t_2\}^p w^{B'})}{l^{(t_2', q), B} := v' \,/\, \mu \xrightarrow{PC} \Uparrow pq}$$

R-Cast-Sub
$$\frac{w_2 \xrightarrow{s_1 \Leftarrow s_2}_p w_1 \qquad B \sqsubseteq B_1}{\{s_1^{B_1} \Leftarrow s_2^{b_2}\}^p w_2^B \,/\, \mu \xrightarrow{PC} w_1^{B \sqcup B_1} \,/\, \mu}$$

R-Cast-To-Dyn
$$\frac{w_2 \xrightarrow{s_1 \Leftarrow s_2}_p w_1}{\{s_1^? \Leftarrow s_2^{b_2}\}^p w_2^B \,/\, \mu \xrightarrow{PC} w_1^B \,/\, \mu}$$

Figure 6. Semantics: Casts, failure, and propagation.

L value in a high context. If the assignment succeeds, then the blame labels are joined because either the current update or a preceding one may be responsible for a later mismatch.

D. Casts

The reduction rules in Fig. 6 define failed reductions and the reduction of successful casts. The rules *R-Ctxt-Fail* and *R-Ctxt-Protect-Fail* propagate security violations through evaluation contexts all the way to the top level, thus assuring that program execution stops.

Failures originate from attempts to expose secure information by downgrading the security level of a value directly or by overwriting a low security value under a high pc. Rule *R-Cast-Sub-Fail* covers the former case by detecting the mismatch of the cast's target type and the run-time security level on its value argument. Rule *R-Asgn-NSU-Fail* avoids illegal upgrades on the heap. This rule enforces the NSU semantics [3]. It checks the current pc against the security level of the memory cell's current content that is to be overwritten. The blame labels are joined.

A cast can be successfully reduced if the top-level annotation of its target type admits the run-time security level of its argument value (rules *R-Cast-Sub* and *R-Cast-To-Dyn*). Additionally there may be sub-casts to consider, which are propagated to the sub-components of the result value. Figure 7 defines the cast propagation $w \xrightarrow{s' \Leftarrow s}_p w'$. It rewrites a raw value w that has raw type s to a raw value w' such that it admits raw type s'. The blame label p indicates the cast that initiated the propagation (cf. rules *R-Cast-Sub* and *R-Cast-To-Dyn*).

Nothing happens to an integer constant or a unit value. To reflect the cast of a reference's content type from t_1 to t_1', the propagation updates the access type of a pointer accordingly.

The most interesting case of cast propagation are function casts where a cast lambda abstraction is rewritten to admit the desired target function type. The handling of conversions for parameter and result types is analogous to that employed by other systems with gradual typing [29]: The body is wrapped in an abstraction and applied to the argument after casting the target argument type to the source argument type. The result of the application is then cast to the target result type. As a minor adjustment for the current security setting, the inner lambda abstraction obtains security level L such that it can be typed under any pc.

It remains to consider the conversion of the program counter security level of the abstraction's body e. The target pc' may not admit the e as it was typed under an unrelated pc. The cast propagation rule for functions of Fig. 7 maintains type safety in such a situation by replacing the original body e with the guard cast expression $\{[\,]pc' \Leftarrow [\,]pc\}^p e$.

Example 1. To illustrate the purpose of guard casts consider the following well-typed cast:

$$\{(s^{\mathsf{L}} \xrightarrow{\mathsf{H}} \mathtt{unit})^{\mathsf{L}} \Leftarrow (s^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathtt{unit})^{\mathsf{L}}\}^p (\lambda^{\mathsf{L}} x.\, l^{(s^{\mathsf{L}}, q), \mathsf{L}} := x)$$

By the rules of Fig. 3, the types $(s^{\mathsf{L}} \xrightarrow{\mathsf{H}} \mathtt{unit})^{\mathsf{L}}$ and $(s^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathtt{unit})^{\mathsf{L}}$ are compatible. However, the function's body $l^{(s^{\mathsf{L}}, q), \mathsf{L}} := x$ is a low-security assignment and cannot be typed under the high-security target pc. In this example type safety under the high-security pc can be restored by modifying the assignment in the body to have a high security write effect: $l^{(s^{\mathsf{H}}, q), \mathsf{L}} := x$. A guard cast performs such type preserving adjustments.[4]

The guard cast related typing rules, given in Fig. 8, are designed to ensure type preservation until the guard cast can perform the adjustments. Rule *T-GuardCast* converts freely between pcs in a typing derivation. It allows the cast propagation rule for functions in Fig. 7 to preserve typing under the target pc. Rule *T-FunCast* casts the pc of a function like a regular cast, but without having to consider parameter and result types. The rule for pointer casts, *T-RefCast*, converts the top-level type annotation of the content of a reference. The additional constraint $pc_2 \sqsubseteq b$ is a technical restriction that helps to maintain an invariant needed in a preservation proof (Sec. VI).

Fig. 9 shows the interesting cases for reductions related to guard casts. Fig. 8 contains the corresponding typing

[4]In principle such adjustments to function bodies could be performed by a meta function in the rule for function propagation. To streamline the proofs, we choose to integrate the adjustments into the reduction relation.

$\boxed{w \xrightarrow{s' \Leftarrow s}_p w'}$

$$k \xrightarrow{\texttt{int} \Leftarrow \texttt{int}}_p k \qquad () \xrightarrow{\texttt{unit} \Leftarrow \texttt{unit}}_p () \qquad l^{(t_1,q)} \xrightarrow{\texttt{ref}\ \ t_1' \Leftarrow \texttt{ref}\ \ t_1}_p l^{(t_1',q)}$$

$$\lambda x.\, e \xrightarrow{t_1' \overset{pc'}{\to} t_2' \ \Leftarrow t_1 \overset{pc}{\to} t_2}_p \lambda x.\, \{t_2' \Leftarrow t_2\}^p(\lambda^{\mathsf{L}} x.\, \{[]pc' \Leftarrow []pc\}^p e)\ \{t_1 \Leftarrow t_1'\}^p x$$

Figure 7. Semantics: Cast propagation.

$\boxed{e \,/\, \mu \xrightarrow{PC} r}$ cont.

R-GuardCast-New
$$\{[]pc' \Leftarrow []pc\}^p \texttt{new}^{s^b,B} e \,/\, \mu \xrightarrow{PC} \texttt{new}^{s^{b \sqcup pc'},B} \{[]pc' \Leftarrow []pc\}^p e \,/\, \mu$$

R-GuardCast-Asgn
$$\{[]pc' \Leftarrow []pc\}^p (e_1 := e_2) \,/\, \mu \xrightarrow{PC} (\{\texttt{ref}\ \ pc' \Leftarrow pc\}^p \{[]pc' \Leftarrow []pc\}^p e_1) := (\{[]pc' \Leftarrow []pc\}^p e_2) \,/\, \mu$$

R-GuardCast-App
$$\{[]pc' \Leftarrow []pc\}^p (e_1\ e_2) \,/\, \mu \xrightarrow{PC} (\{* \overset{pc' \Leftarrow pc}{\to} *\}^p \{[]pc' \Leftarrow []pc\}^p e_1)\ (\{[]pc' \Leftarrow []pc\}^p e_2) \,/\, \mu$$

R-GuardCast-GuardCast
$$\{[]pc_1 \Leftarrow []pc_2\}^p \{[]pc_1' \Leftarrow []pc_2'\}^q e \,/\, \mu \xrightarrow{PC} \{[]pc_1 \Leftarrow []pc_2'\}^{pq} e \,/\, \mu$$

R-FunCast
$$\{* \overset{pc_1 \Leftarrow pc_2}{\to} *\}^p \lambda^B x.\, e \,/\, \mu \xrightarrow{PC} \lambda^B x.\, \{[]pc_1 \Leftarrow []pc_2\}^p e \,/\, \mu$$

R-RefCast
$$\{\texttt{ref}\ \ pc_1 \Leftarrow pc_2\}^p l^{(s^b,q),B} \,/\, \mu \xrightarrow{PC} l^{(s^{b \sqcup pc_1},pq),B} \,/\, \mu$$

Figure 9. Reduction rules for guard casts (interesting cases).

T-GuardCast
$$\frac{pc; \Gamma; M \vdash e : t}{pc'; \Gamma; M \vdash \{[]pc' \Leftarrow []pc\}^p e : t}$$

T-FunCast
$$\frac{pc; \Gamma; M \vdash e : (t_1 \overset{pc_2}{\to} t_2)^b}{pc; \Gamma; M \vdash \{* \overset{pc_1 \Leftarrow pc_2}{\to} *\}^p e : (t_1 \overset{pc_1}{\to} t_2)^b}$$

T-RefCast
$$\frac{pc'; \Gamma; M \vdash e : \texttt{ref}^{\,b'}\ s^b \qquad pc_2 \sqsubseteq b}{pc'; \Gamma; M \vdash \{\texttt{ref}\ \ pc_1 \Leftarrow pc_2\}^p e : \texttt{ref}^{\,b'}\ s^{b \sqcup pc_1}}$$

Figure 8. Typing rules for guard casts and pointer casts.

rules. The cases not shown in Fig. 9 either drop the guard cast at values or propagate it to the subexpressions. Rule *R-GuardCast-New* augments the allocation type of a `new` expression, allowing it to allocate a reference at a sufficiently high security level to be permitted by the target pc. Rule *R-GuardCast-Asgn* applies an pointer cast to the updated reference making the result well-typed under the target pc by typing rules *T-RefCast* and *T-Asgn*. Once the reference is fully evaluated, rule *T-RefCast* adjusts the pointer's access type analogously to Example 1. The situation for function application is similar: rule *R-GuardCast-App* needs to introduce a function guard cast to the function being applied. Once the function is evaluated, it converts the pc of the function's body to a sufficiently high level with rule *R-FunCast*. If two guard casts collide they are joined by rule *R-GuardCast-GuardCast*. For well-typed expressions, the target pc of the first cast, pc_1' and the source pc of the second cast, pc_2 are related contravariantly, i.e. $pc_2 \sqsubseteq pc_1'$, due to rules *T-GuardCast* and *T-Sub*.

VI. Type Soundness

Type soundness for ML-GS is established in the usual way by proving type preservation and progress. As these results need to be proven for configurations, typing must be augmented with a heap typing judgment $M \vdash \mu$ defined in Fig. 10. It characterizes well-typed heaps using the address environment M. Its single rule, *T-Heap*, checks that the typing information in the cast values is consistent. The auxiliary judgment $M; l \vdash v : t$ requires that the value v is typed with the current type t and that t is compatible with the type that M binds to l. Requiring compatibility ensures, together with the *R-Loc* typing rule, that reduction

T-Heap-Aux

$$\frac{t \sim M(l) \qquad \mathsf{L}; \text{-}; M \vdash v : t}{M; l \vdash v : t}$$

T-Heap

$$\frac{\mathrm{dom}(M) = \mathrm{dom}(\mu) \qquad \forall (l \mapsto \{t\}^q v) \in \mu.\ M; l \vdash v : t}{M \vdash \mu}$$

Figure 10. Heap typing.

rule *R-Deref* constructs a valid cast when dereferencing a pointer.

Theorem 1 (Preservation)**.** *Let PC be a run-time program counter security level and pc a typing program counter security level such that $PC \sqsubseteq pc$. If $pc; \text{-}; M \vdash e : t$ and $M \vdash \mu$ and $e \,/\, \mu \xrightarrow{PC} e' \,/\, \mu'$ then there exists M' such that $pc; \text{-}; M, M' \vdash e' : t$ and $M, M' \vdash \mu'$.*

Proof: By induction on the derivation of $e \,/\, \mu \xrightarrow{PC} e' \,/\, \mu'$. The proof relies on the usual substitution and weakening results for typing contexts. As in Pottier and Simonet's work [21], the pc of a typing derivation can be decreased arbitrarily. Further details are given in Appendix A. ∎

Progress characterizes the possible outcomes of the one-step evaluation of a typed program.

Theorem 2 (Progress)**.** *If $pc; \text{-}; M \vdash e : t$ and $M \vdash \mu$ then, for all $PC \sqsubseteq pc$ either (i) e is a value, (ii) there exists p such that $e \,/\, \mu \xrightarrow{PC} \Uparrow p$, or (iii) there exist μ' and e' such that $e \,/\, \mu \xrightarrow{PC} e' \,/\, \mu'$.*

Proof: By induction on the derivation of $pc; \text{-}; M \vdash e : t$. Details are given in Appendix B. ∎

The statement of progress shows that even a well-typed program may fail with an unsafe cast. The following definition of an unsafe program classifies a proper subset of configurations for which evaluation can result in a failure. The complement of this subset are *safe configurations* that cannot fail.

Definition 1. Let p be a blame label and M an address environment. Configuration $e \,/\, \mu$ is *unsafe for p under M* iff

- there is a casts $\{t \Leftarrow t'\}^q e$ that occurs in e or μ where $q \subseteq p$ and $t' \not\prec t$,
- there is a function guard casts $\{* \overset{pc \Leftarrow pc'}{\rightarrow} *\}^q e'$, pointer casts $\{\texttt{ref}\ \ pc \Leftarrow pc'\}^q e'$ or guard casts $\{[]pc \Leftarrow []pc'\}^q e'$ that occurs in e or μ where $q \subseteq p$ and $pc \not\sqsubseteq pc'$,
- there is a pointer $l^{(t,q),b}$ that occurs in e or μ where $q \subseteq p$ and $t \neq M(l)$, or
- there is a heap binding $(l \mapsto \{t'\}^q v)$ that occurs in μ where $q \subseteq p$ and $t' \neq M(l)$.

Thus, for a cast to be unsafe it has to violate the subtyping relation which means it performs a conversion that is not allowed by the static fragment of the type system. For pointers and heap bindings to be unsafe, their current types and access types must differ from the type that is assumed by the static address environment M.

Well-typed programs reduce to unsafe configurations only if they were originally unsafe.

Theorem 3. *Let p be a blame label, M and M' address environments, $e \,/\, \mu$ and $e' \,/\, \mu'$ configurations. Let furthermore $e \,/\, \mu$ be well-typed under address environment M and $e' \,/\, \mu'$ well typed under address environment M, M'. It holds that if $e' \,/\, \mu'$ is unsafe for p under M, M' and $e \,/\, \mu \xrightarrow{PC} e' \,/\, \mu'$, then $e \,/\, \mu$ is unsafe for p under M.*

Proof: By induction on the derivation of $e \,/\, \mu \xrightarrow{PC} e' \,/\, \mu'$. In the *R-Cast-*$*$ cases, check that cast propagation decomposes casts in accordance to the subtyping rules; a configuration that is not unsafe cannot produce an unsafe one. The other interesting cases are *R-New*, *R-Deref*, *R-Asgn*, and *R-Ref-Cast*. In case *R-New*, where the heap is extended, check that the resulting fresh reference is never unsafe for any blame label. In case *R-Deref*, if the cast $\{t'_2 \Leftarrow t_2\}^{pq} v$ is the cause for unsafety, it holds that $t'_2 \not\prec t_2$ and in particular $t'_2 \neq t_2$. It follows that the access type of the pointer and the current type of heap binding can never agree on a common type $M(l)$ and therefore the heap binding or the pointer makes the original configuration unsafe. In case *R-Asgn*, when the updated heap binding is the cause of unsafety, we have $B_1 = B_1 \sqcup PC \sqcup B$ because of the NSU check. There are two possible cases: In case $b_1 \neq b_2 \sqcup PC \sqcup B$ and thus, $b_1 \sqcup PC \sqcup B \neq b_2 \sqcup PC \sqcup B$ we have $b_1 \neq b_2$ and either the original heap binding or the pointer does not agree with M. In case $b_1 = b_2 \sqcup PC \sqcup B$ the original heap binding is already unsafe. In the case of rule *R-RefCast* the technical constraint $pc_2 \sqsubseteq b$ in the corresponding typing rule *T-RefCast* ensures, together with the assumed safety of the cast expression, that $pc_1 \sqsubseteq pc_2 \sqsubseteq b$. Therefore $b \sqcup pc_1 = b$ and it follows that an unsafe pointer in the result expressions is also present in the original configuration. ∎

The cause for a failing reduction is always an unsafe configuration:

Theorem 4. *Let p be a blame label and M an address environment and $e \,/\, \mu$ a configuration. Further, let $pc; \text{-}; M \vdash e : t$. If $e \,/\, \mu \xrightarrow{PC} \Uparrow p$ then $e \,/\, \mu$ is unsafe for p.*

Proof: By induction on the derivation of $e \,/\, \mu \xrightarrow{PC} \Uparrow p$. ∎

From the Theorems 3 and 4 it follows that evaluating a safe configuration will not fail.

Corollary 1. *Let $e \,/\, \mu$ be a closed, well-typed configuration under M and p a blame label. If $e \,/\, \mu$ is safe for all $q \subseteq p$*

$$\begin{array}{lll} \textit{Raw Values} & & w_{\mathcal{L}} ::= \ldots \mid \mathfrak{H} \\ \textit{El. Contexts} & & V ::= [\,]\ v \mid \mathtt{new}^{t,B}[\,] \\ & & \mid\, !\,[\,] \mid [\,] := v \end{array}$$

R-High-Elim

$$V[\mathfrak{H}^{\mathsf{L}}] \,/\, \mu \xrightarrow{\mathsf{L}} \mathfrak{H}^{\mathsf{L}} \,/\, \mu$$

R-High-Deref

$$\frac{l \notin \mathrm{dom}(\mu)}{!\ l^{(t_2',q),B} \,/\, \mu \xrightarrow{\mathsf{L}} \mathfrak{H}^{\mathsf{L}} \,/\, \mu}$$

R-High-Asgn

$$\frac{l \notin \mathrm{dom}(\mu) \qquad \mu' = \mu, (l \mapsto \{t_2'\}^q w^{\mathsf{L}})}{l^{(t_2',q),\mathsf{L}} := w^{\mathsf{L}} \,/\, \mu \xrightarrow{\mathsf{L}} ()^{\mathsf{L}} \,/\, \mu'}$$

Figure 11. Extensions for projection syntax and semantics.

under M *then* $e \,/\, \mu \overset{PC}{\not\longrightarrow} \Uparrow p$.

It is easy to realize that with this result and with Theorems 1 and 2 the safe subset of ML-GS enjoys the usual type soundness property.

VII. Non-Interference

We prove non-interference for ML-GS by adapting a proof technique due to Li and Zdancewic [19]. A projection function first removes all security features of a program and replaces high-security values with an opaque placeholder value, $\mathfrak{H}$. The semantics is extended with elimination rules for $\mathfrak{H}$ values that simply result in $\mathfrak{H}$, thus hiding high-security computations. Then we show the *projection theorem*: all ML-GS executions have corresponding projected executions. Non-Interference is a corollary of the projection theorem together with type soundness.

A. Projections

The extensions for the projected language, here called ML-GS$_L$, are given in Fig. 11. We write $e_{\mathcal{L}}$, $v_{\mathcal{L}}$, and $w_{\mathcal{L}}$ to distinguish expressions, values and raw values of ML-GS$_L$ from the respective ML-GS forms. Raw values are extended with the value $\mathfrak{H}$. The additional reduction rules ensure that the flow of high-security information is approximated correctly. Rule *R-High-Elim* reduces to $\mathfrak{H}$ on any attempt to eliminate a $\mathfrak{H}$ value. Rules *R-High-Deref* and *R-High-Asgn* cover the cases where the projection removed high-security values from the heap whose addresses were in scope of the projected expression.

The projection function $\mathcal{L}(\cdot)$ transforms an ML-GS expression into an ML-GS$_L$ expression thereby abstracting all high-security information to $\mathfrak{H}$ and removing all security features. Fig. 12 defines the interesting cases of $\mathcal{L}(\cdot)$; the remaining cases propagate $\mathcal{L}(\cdot)$ homomorphically. The affected expressions are high-security values, protection by a high security level, high-security memory allocation, and casts to a high-security type. The accompanying type

$\boxed{\lceil t \rceil_{\mathsf{L}}}$

$$\lceil \mathtt{int}^b \rceil_{\mathsf{L}} = \mathtt{int}^{\mathsf{L}} \quad \lceil \mathtt{unit}^b \rceil_{\mathsf{L}} = \mathtt{unit}^{\mathsf{L}}$$
$$\lceil \mathtt{ref}^{\,b}\ t \rceil_{\mathsf{L}} = \mathtt{ref}^{\,\mathsf{L}}\ \lceil t \rceil_{\mathsf{L}}$$
$$\lceil (t_1 \xrightarrow{pc} t_2)^b \rceil_{\mathsf{L}} = (\lceil t_1 \rceil_{\mathsf{L}} \xrightarrow{\mathsf{L}} \lceil t_2 \rceil_{\mathsf{L}})^{\mathsf{L}}$$

$\boxed{\mathcal{L}(e)}$ $\boxed{\mathcal{L}(\mu)}$

$$\begin{array}{rr} \mathcal{L}(w^{\mathsf{H}}) = \mathfrak{H}^{\mathsf{L}} & \mathcal{L}(()^{\mathsf{L}}) = ()^{\mathsf{L}} \\ \mathcal{L}(\mathtt{prot}^{\mathsf{H}} e) = \mathfrak{H}^{\mathsf{L}} & \mathcal{L}(k^{\mathsf{L}}) = k^{\mathsf{L}} \\ \mathcal{L}(\mathtt{new}^{t,\mathsf{H}} e) = \mathfrak{H}^{\mathsf{L}} & \mathcal{L}(\lambda^{\mathsf{L}} x.\, e) = \lambda^{\mathsf{L}} x.\, \mathcal{L}(e) \\ \mathcal{L}(\{s^{\mathsf{H}} \Leftarrow t_2\}^p e) = \mathfrak{H}^{\mathsf{L}} & \mathcal{L}(l^{(t',p),\mathsf{L}}) = l^{(\lceil t' \rceil_{\mathsf{L}},p),\mathsf{L}} \end{array}$$

$$\mathcal{L}(x) = x$$
$$\mathcal{L}(\{s^b \Leftarrow t_2\}^p e) = \{\lceil s^b \rceil_{\mathsf{L}} \Leftarrow \lceil t_2 \rceil_{\mathsf{L}}\}^p \mathcal{L}(e) \quad \text{if } b \neq \mathsf{H}$$
$$\mathcal{L}(\{[\,]pc' \Leftarrow [\,]pc\}^p e) = \mathcal{L}(e)$$
$$\mathcal{L}(\{* \overset{pc' \Leftarrow pc}{\rightarrow} *\}^p e) = \mathcal{L}(e)$$
$$\mathcal{L}(\{\mathtt{ref}\ \ pc' \Leftarrow pc\}^p e) = \mathcal{L}(e)$$
$$\mathcal{L}(\mathtt{new}^{t,\mathsf{L}} e) = \mathtt{new}^{\lceil t \rceil_{\mathsf{L}},\mathsf{L}} \mathcal{L}(e)$$

$$\mathcal{L}(\text{-}) = \text{-}$$
$$\mathcal{L}(\mu, (l \mapsto \{t_2\}^p w^{\mathsf{H}})) = \mathcal{L}(\mu)$$
$$\mathcal{L}(\mu, (l \mapsto \{t_2\}^p w^{\mathsf{L}})) = \mathcal{L}(\mu), (l \mapsto \{\lceil t_2 \rceil_{\mathsf{L}}\}^p w^{\mathsf{L}})$$

Figure 12. Projections (interesting cases).

projection $\lceil \cdot \rceil_{\mathsf{L}}$ replaces all type annotations by L and thus transforms all other casts into identities.

On heaps, $\mathcal{L}(\cdot)$ removes all high-security cells which reflects our assumption that a difference in the size of the high-security part of the heap is not a security leak. Our non-interference proof relies on the removal, as ML-GS cannot guarantee that all possible high-security dependent executions allocate the same amount of memory.

In summary, as the *R-High-Elim* rule ensures the correct flow of hidden high-security information, an expression $\mathcal{L}(e)$ can be executed without security concerns.

B. Non-Interference Proof

The projection theorem is the main stepping stone towards proving non-interference. We need some lemmas. First, check that heap updates under high-security pcs are invisible under projection.

Lemma 1. *If* $e \,/\, \mu \xrightarrow{\mathsf{H}} e' \,/\, \mu'$ *then* $\mathcal{L}(\mu) = \mathcal{L}(\mu')$.

Proof: By induction on the derivation of $e \,/\, \mu \xrightarrow{\mathsf{H}} e' \,/\, \mu'$. In the case *R-New*, a high-security binding is created on the heap due to the high-security rely on the fact that projections throws away all high-security bindings. ■

Using this lemma the projection theorem follows:

Lemma 2. *If* $e \,/\, \mu \xrightarrow{L} e' \,/\, \mu'$ *then* $\mathcal{L}(e) \,/\, \mathcal{L}(\mu) \xrightarrow{L}{}^{*} \mathcal{L}(e') \,/\, \mathcal{L}(\mu')$.

Proof: By induction on the derivation of $e \,/\, \mu \xrightarrow{L} e' \,/\, \mu'$. Use lemma 1 in case *R-Ctxt-Protect*. ∎

Theorem 5 (Projection)**.** *If* $e \,/\, \mu \xrightarrow{L}{}^{*} e' \,/\, \mu'$ *then* $\mathcal{L}(e) \,/\, \mathcal{L}(\mu) \xrightarrow{L}{}^{*} \mathcal{L}(e') \,/\, \mathcal{L}(\mu')$.

With the projection theorem and type soundness, non-interference follows. We restrict ourselves here to integer results; pointers work similarly and non-interference for units is trivial. Non-Interference of function values can be shown, similarly as in Austin and Flanagan's work [5], using an equivalence relation based on projection.

Theorem 6 (Non-Interference)**.** *Let* v_i, $i \in \{1, 2\}$ *be two arbitrary, high-security values with respective typings* $\mathsf{L}; \text{-}; M \vdash v_i : s^{\mathsf{H}}$. *If* $\mathsf{L}; x : s^{\mathsf{H}}; M \vdash e : \texttt{int}^{L}$ *and* $e[x \mapsto v_i] \,/\, \mu \xrightarrow{L}{}^{*} v_i' \,/\, \mu_i$ *then* $v_1' = v_2'$.

Proof: By the projection theorem it holds that $\mathcal{L}(e[x \mapsto v_i]) \,/\, \mathcal{L}(\mu) \xrightarrow{L}{}^{*} \mathcal{L}(\mu_i) \,/\, \mathcal{L}(v_i')$. By type soundness, $v_i' = k_i^{\mathsf{L}}$ are low-security integers. With $\mathcal{L}(e[x \mapsto v_1]) = \mathcal{L}(e[x \mapsto v_2])$ and deterministic evaluation for ML-GS$_L$ it follows that $\mathcal{L}(k_1^{\mathsf{L}}) = \mathcal{L}(k_2^{\mathsf{L}})$. Projection is injective for low-security values and therefore the result follows. ∎

VIII. Discussion

The blame labels of ML-GS are sets of blame ids. In a concrete implementation, a programmer (or rather the IDE or compiler) would label all casts with distinct singleton sets of blame ids. The blame label attached to a blame exception gives the programmer a hint where to look for the bug that caused the security leak. This hint should be as precise as possible. However, currently the blame labels of ML-GS exceptions are very imprecise: Corollary 1 only guarantees that a non-empty subset of the blame label signaled by the exception is responsible for the failure.

The cause of this imprecision are the unconditional joins of blame labels in rules like *R-Deref*. With the type information that is currently available at run time it is not possible to determine which of the blame labels, that on the access type of the pointer or that on the current type of the heap binding, is the real culprit for a potential blame exception later on. As an example, consider the following configuration:

$$!\, l^{(s^{\mathsf{L}},p),\mathsf{L}} \,/\, (l \mapsto \{s^{\mathsf{H}}\}^{q} v) \tag{1}$$

The dereferencing would result in a blame exception $\Uparrow pq$ due to the low-security access of the high-security content of the address l. The following two programs both reduce to configuration (1).

```
1 p1 =
2  let l = new^L v' in
3  ({s^H <= s^L}[q] l) := v;
4  !{s^L <= s^L}[p]l
5
6 p1 =
7  let l = new^H v' in
8  ({s^H <= s^H}[q] l) := v;
9  !{s^L <= s^H}[p]l
```

In program `p1` the cast q is at fault whereas the cast p is the culprit in program `p2`. This distinction cannot be reconstructed in (1) with the current run-time annotations.

However, annotating heap bindings and pointers with the original, allocation-time type of the pointer allows us to recover the distinction.

$$\textit{Values}\ \ v ::= \ldots \mid l^{(t \Leftarrow t,p),B}$$
$$\textit{Heaps}\ \ \mu ::= \cdot \mid (l \mapsto \{t \Leftarrow t\}^{p} v)$$

The left of the type annotations in the new pointer and heap syntax are the access and current type respectively, as before. The right annotation is the *static type* which always coincides with the type in the address environment. The static type is fixed at allocation-time and, for a particular address, the static type is the same for the heap binding and all pointers. With a straightforward extension of the semantics, `p1` now reduces to

$$!\, l^{(s^{\mathsf{L}} \Leftarrow s^{\mathsf{L}},p),\mathsf{L}} \,/\, (l \mapsto \{s^{\mathsf{H}} \Leftarrow s^{\mathsf{L}}\}^{q} v)$$

where the static type is s^{L} due to the low-security initialization. It is now easy to decide that cast q is to blame, as static type and current type differ, and it would suffice to track label q from here on.

IX. Related Work

There are many proposals for static program analysis of information flow. Most are inspired by the lattice-based model of the Dennings [8], [9], which categorizes information according to security levels. On that basis, Volpano, Smith, and Irvine [28] were the first to construct a type system that analyzes confidentiality for a simple imperative language, followed by a flurry of subsequent work. Later work, relevant to our paper, extends their principles to integrity checking and to higher-order languages [14], [21]. Sabelfeld and Myers give a comprehensive overview [23]. The approaches based on type systems augment types with security annotations that over-approximate the confidentiality level of the information contained in a value. These systems rule out programs where a high-confidential input may leak to a low-confidential output.

As an alternative for static analysis, a number of authors have examined dynamic information flow analysis. The basic approach extends the run-time system with a monitor or augments values with security levels so that potential security violations can be detected during the execution of a program [24]. The monitoring modifies the observable output of the program to guarantee non-interference.

The main difficulty of dynamic analysis is the taming of implicit flows, which was deemed infeasible for some time and gave rise to hybrid approaches [12]. Recent advances have developed a range of techniques with increasingly good results. Sabelfeld and Russo [24] demonstrate a runtime monitor that guarantees termination-insensitive noninterference (TINI) and is more permissive than a flow-insensitive analysis. Austin and Flanagan proposed the no-sensitive-upgrade policy [3], improved that to the permissive upgrade policy [4], and subsumed both by faceted execution [5] (all guaranteeing TINI), which approximates secure multi-execution (SME), where a program is executed multiple times, once for each security level [10]. When SME executes a level, the information that is confidential for that level is overridden with a default value.

Russo and Sabelfeld [22] compare flow-sensitive static security analysis with an axiomatically described family of dynamic analyses and prove that their results are incomparable. More accurately, they prove that a dynamic analysis which is strictly stronger than the flow-sensitive analysis of Hunt and Sands [17] is not possible. Russo and Sabelfeld further suggest a hybrid analysis that processes a high-security conditional by executing one branch and statically analyzing the other [12]. This analysis is more permissive than the flow-sensitive static analysis. Their results support the need for combining static and dynamic analysis like our proposal. As our base analysis is flow-insensitive, Sabelfeld and Russo's earlier result [24] shows that our dynamic analysis is strictly more permissive than our static analysis, so our combination is useful. Furthermore, we believe our approach is complementary because even a flow-sensitive static analysis cannot cope with our example from Sec. II.

TINI, as established for our system, does not provide perfect security. Askarov and coworkers [2] generally criticize the notion of termination insensitive non-interference and suggest alternative definitions. Non-interference is also insufficient in the presence of timing attacks as discussed by Kashyap and coworkers [18].

Gradual typing [25], [26] originates from the desire to execute dynamically typed programs efficiently and builds on earlier work on dynamic typing [15] and soft typing [7]. Cast expressions manifest typing information locally so that more efficient, untagged data representations can be employed. This approach has also proven useful in the integration of dynamically typed scripting languages with typed languages [6], [20], [30], where the primary goal is improved maintainability and interoperability. Wadler and Findler [29] characterize the interaction precisely with their blame theorem, which identifies safe parts of a program that never give rise to type errors.

Gradual typing for mutable data has only been considered by two papers [16], [27]. Both papers employ reference coercions with two components, one for reading and one for writing, and they perform coercion simplification that can fail early, before the coercion is applied. In contrast, our reference coercions have one component and behave differently: we can always write to the reference according to its current type, even if that type is the result of a cast. Read operations may lead to failure. In the other approaches, read and write operations may fail. We do not check casts for early failure because even a cast from $\texttt{ref}^{\,\mathsf{H}}$ to $\texttt{ref}^{\,\mathsf{L}}$ is acceptable, if the next operation on the reference is a write, which is enforced by our semantics. A variant of gradual security typing that behaves like the two papers is feasible, but we believe that our semantics fits better with patterns used in secure software construction.

Siek and coworkers [27] emphasize precise blame tracking whereas we deliberately keep that part of the system simple. Furthermore, their system does not keep track of effects, which is indispensable for handling implicit flows in our system.

The combination of gradual typing and security analysis has been considered [11], but only in the context of the pure lambda calculus, whereas we consider an ML core language with references. Our approach to modeling the calculus and to proving non-interference are quite different and are extensible to a realistic language with arbitrary effects.

X. Conclusion

We apply the ideas of gradual typing to an annotated type system for information flow control. The construction of the gradual system is straightforward in a pure lambda calculus setting, but poses significant challenges when performed for an ML core language with references as we do. While the statically typed part can be adapted from previous work [21], the design of the cast operators and the integration of dynamic information flow techniques in the operational semantics are novel to our work. The particular challenge in our system is the design of casts on references. They are not restricted by the subtyping relation, they never fail, and they always result in a reference that is ready for writing at the target type of the cast. We further demonstrate that the gradual type system is independent from the enforcement strategy used in the untyped part of a program. The formalization in the paper uses the NSU strategy, but we have also worked out the details for the FE strategy, which is only sketched here.

We envisage ML-GS to be useful in contexts where security requirements change or where language features prohibit the use of static analysis throughout. For example, ML-GS can integrate manually security-checked code in a typed setting, thus creating safe, but dynamically checked, regions inside of statically checked code. Likewise, the integration of statically checked regions in dynamically checked code is also possible. These regions can be enlarged or shrunk according to security and robustness requirements.

An extension of the blame theorem [29] can be stated and proved for ML-GS, but it is omitted to conserve space.

Future work considers an inference for placing cast expressions and the addition of ML-polymorphism along the lines of Pottier and Simonet. It should also be possible to combine gradual security typing with plain gradual typing, but the focus of the present work is on the security aspect.

Acknowledgment

Thanks to Joshua Guttman for his extensive, thoughtful comments on draft versions of this paper, which helped to improve the presentation considerably.

References

[1] A. Ahmed, R. B. Findler, J. Matthews, and P. Wadler. Blame for all. In *Proceedings for the 1st workshop on Script to Program Evolution*, pages 1–13, Genova, Italy, 2009. ACM.

[2] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In S. Chong and D. A. Naumann, editors, *PLAS*, pages 113–124, Dublin, Ireland, June 2009. ACM.

[4] 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*, PLAS '10, pages 3:1–3:12, New York, NY, USA, 2010. ACM.

[5] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proc. 39th ACM Symp. POPL*, pages 165–178, Philadelphia, USA, Jan. 2012. ACM Press.

[6] G. M. Bierman, E. Meijer, and M. Torgersen. Adding dynamic types to C#. In T. D'Hondt, editor, *ECOOP*, volume 6183 of *LNCS*, pages 76–100, Maribor, Slovenia, 2010. Springer.

[7] R. Cartwright and M. Fagan. Soft typing. In *Proc. PLDI '91*, pages 278–292, Toronto, Canada, June 1991. ACM.

[8] D. Denning. A lattice model of secure information flow. *Comm. ACM*, 19(5):236–242, 1976.

[9] D. Denning and P. Denning. Certification of programs for secure information flow. *Comm. ACM*, 20(7):504–513, 1977.

[10] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *IEEE Symposium on Security and Privacy*, pages 109–124, Berkeley/Oakland, California, USA, May 2010. IEEE Computer Society.

[11] T. Disney and C. Flanagan. Gradual information flow typing. In *STOP 2011*, 2011.

[12] G. L. Guernic, A. Banerjee, T. P. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In M. Okada and I. Satoh, editors, *ASIAN*, volume 4435 of *LNCS*, pages 75–89. Springer, 2006.

[13] D. Hedin and A. Sabelfeld. A perspective on information-flow control. In *2011 Marktoberdorf Summer School*. IOS Press, 2011.

[14] N. Heintze and J. G. Riecke. The SLam calculus: Programming with security and integrity. In L. Cardelli, editor, *Proc. 25th ACM Symp. POPL*, pages 365–377, San Diego, CA, USA, Jan. 1998. ACM Press.

[15] F. Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22:197–230, 1994.

[16] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *Trends in Functional Programming (TFP)*, 2007.

[17] S. Hunt and D. Sands. On flow-sensitive security types. In S. Peyton Jones, editor, *Proc. 33rd ACM Symp. POPL*, pages 79–90, Charleston, South Carolina, USA, Jan. 2006. ACM Press.

[18] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 413–428, Washington, DC, USA, 2011. IEEE Computer Society.

[19] P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19):1974–1994, 2010.

[20] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *ACM TOPLAS*, 31:12:1–12:44, Apr. 2009.

[21] F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS*, 25(1):117–158, Jan. 2003.

[22] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *CSF*, pages 186–199. IEEE Computer Society, 2010.

[23] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[24] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In A. Pnueli, I. Virbitskaite, and A. Voronkov, editors, *Ershov Memorial Conference*, volume 5947 of *Lecture Notes in Computer Science*, pages 352–365. Springer, 2009.

[25] J. Siek and W. Taha. Gradual typing for objects. In E. Ernst, editor, *21st ECOOP*, volume 4609 of *LNCS*, pages 2–27, Berlin, Germany, July 2007. Springer.

[26] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Sept. 2006.

[27] J. G. Siek, M. M. Vitousek, and S. Bharadwaj. Gradual typing for mutable objects. http://ecee.colorado.edu/~siek/gtmo.pdf, Dec. 2012.

[28] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):1–21, 1996.

[29] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In G. Castagna, editor, *Proc. 18th ESOP*, volume 5502 of *LNCS*, pages 1–16, York, UK, Mar. 2009. Springer-Verlag.

[30] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In J. Palsberg, editor, *Proc. 37th ACM Symp. POPL*, pages 377–388, Madrid, Spain, Jan. 2010. ACM Press.

APPENDIX A.
PROOF OF TYPE PRESERVATION

A. Auxiliary Lemmas

It is straightforward to check that the subtyping relation is well-behaved, in that each well-typed value can be typed by applying the subsumption rule first, followed by the canonical rule for that value (i.e. *T-Abs* is the canonical rule for function abstractions). We will use this fact implicitly in the following.

Lemma 3 (Substitution)**.** *If* $pc; x : t; M \vdash e : t'$ *and* $pc;$ -$; M \vdash v : t$ *then* $pc;$ -$; M \vdash e[x \mapsto v] : t'$.

Proof: By induction on the derivation of $pc; x : t; M \vdash e : t'$. ∎

Lemma 4. *If* $pc;$ -$; M \vdash e : t$ *then for all* $pc' \sqsubseteq pc$ *it holds that* $pc';$ -$; M \vdash e : t$.

Proof: By induction on the derivation of $pc; x : t; M \vdash e : t'$. ∎

Lemma 5. *If* $pc;$ -$; M \vdash e : t$ *then for all* M' *where* $\mathrm{dom}(M)$ *and* $\mathrm{dom}(M')$ *are disjoint, it holds that* $pc;$ -$; M, M' \vdash e : t$.

Lemma 6. *If* $pc;$ -$; M \vdash v : t$ *then for all* pc' *it holds that* $pc';$ -$; M \vdash v : t$.

Proof: By induction on the derivation of $pc;$ -$; M \vdash v : t$. ∎

Lemma 7. *If* $pc;$ -$; M \vdash E[e] : t$ *and* $pc;$ -$; M \vdash e : t'$ *and* $pc;$ -$; M \vdash e' : t'$ *then* $pc;$ -$; M \vdash E[e'] : t$.

Proof: By induction on the derivation of $pc;$ -$; M \vdash E[e] : t$. ∎

Lemma 8. *If* $pc;$ -$; M \vdash w^B : s^b$ *and* $w \xrightarrow{s_1 \Leftarrow s}_p w_1$ *and* $s^b \sim s_1^b$ *then* $pc;$ -$; M \vdash w_1^B : s_1^b$

Proof: By examining the cases of $w \xrightarrow{s_1 \Leftarrow s}_p w_1$ and straightforward type reconstruction, using lemma 4. ∎

B. Proof

Let PC be a run-time security guard and pc a typing security guard such that $PC \sqsubseteq pc$. If $pc;$ -$; M \vdash e : t$ and $M \vdash \mu$ and $e \,/\, \mu \xrightarrow{PC} e' \,/\, \mu'$ then there exists M' such that $pc;$ -$; M, M' \vdash e' : t$ and $M, M' \vdash \mu'$. The proof is an induction of the derivation of $e \,/\, \mu \xrightarrow{PC} e' \,/\, \mu'$.

1) Case R-App: Assumptions:

$$pc;\text{-}; M \vdash (\lambda^B x.\, e)\ v : s^{b \sqcup b'} \tag{2}$$

The heap typing result is immediate.
By (2) and subtyping:

$$pc'; x : t_2'; M \vdash e : s_1^{b_1} \text{ where } pc \sqcup b \sqsubseteq pc' \text{ and } s_1^{b_1} \prec s^{b \sqcup b'} \tag{3}$$

$$pc;\text{-}; M \vdash v : t_2'' \text{ where } t_2'' \prec t_2' \tag{4}$$

By (3), (4), and substitution:

$$pc';\text{-}; M \vdash e[x \mapsto v] : s_1^{b_1} \tag{5}$$

By (5), (3), subtyping and typing rule *T-Protect*:

$$pc;\text{-}; M \vdash \mathtt{prot}^B e[x \mapsto v] : s^{b \sqcup b'} \tag{6}$$

which is the desired result.

2) Case R-Protect: Assumptions

$$pc;\text{-}; M \vdash \mathtt{prot}^B w^{B_1} : s^{B \sqcup b_1} \tag{7}$$

The heap typing result is immediate.
By subtyping and (7):

$$pc \sqcup b;\text{-}; M \vdash w^{B_1} : s_2^{B_1} \text{ and } s_2^{B_1} \prec s^{b_1} \tag{8}$$

By (8) and lemma 6:

$$pc;\text{-}; M \vdash w^{B_1} : s_2^{B_1} \tag{9}$$

By (9), (8) and subtyping:

$$pc;\text{-}; M \vdash w^{B_1 \sqcup B} : s^{B \sqcup b_1} \tag{10}$$

which is the desired result.

3) Case R-New: Assumptions:

$$pc;\text{-}; M \vdash \mathtt{new}^{s^{b_1}, B} w^{B_1} : \mathtt{ref}^{\,b}\ s^{b_1} \tag{11}$$

$$M \vdash \mu \tag{12}$$

By (11):

$$pc;\text{-}; M \vdash w^{B_1} : s^{b_1} \text{ where } PC \sqsubseteq b_1 \tag{13}$$

By (13):

$$pc;\text{-}; M \vdash w^{B_1 \sqcup PC} : s^{b_1} \tag{14}$$

By (14), (12):

$$M, l : s^{b_1} \vdash \mu, (l \mapsto \{t\}^p w^{B_1 \sqcup PC}) \tag{15}$$

which is the desired heap typing result.
By (13) and rule *T-Addr*:

$$pc;\text{-}; M, s^{b_1} \vdash l^{(s^{b_1}, p), B} : \mathtt{ref}^{\,b}\ s^{b_1} \tag{16}$$

which is the desired expression typing result.

4) Case R-Deref: Assumptions:

$$pc; \text{-}; M, l : t_1 \vdash ! \; l^{(s^b, p), B} : s^{b \sqcup b'} \quad (17)$$

$$M, l : t_1 \vdash \mu, (l \mapsto \{t_1'\}^q v) \quad (18)$$

The heap typing result is immediate.
By (17):

$$pc; \text{-}; M, l : t_1 \vdash l^{(s^b, p), B} : \texttt{ref}^{\, b} \; s^{b'} \text{ where } B \sqsubseteq b \text{ and } s^b \sim t_1 \quad (19)$$

By (18) and lemma 6:

$$pc \sqcup B; \text{-}; M, l : t_1 \vdash v : t_1' \text{ where } t_1 \sim t_1' \quad (20)$$

By (19), (20), typing rules *T-Cast* and *T-Protect*, and subtyping:

$$pc; \text{-}; M, l : t_1 \vdash \texttt{prot}^{\, B}\{s^b \Leftarrow t_1'\}^{pq} v : s^b \text{ where } t_1 \sim t_1' \quad (21)$$

Which is the desired result.

5) Case R-Asgn: Assumptions:

$$pc; \text{-}; M, l : t_1 \vdash l^{(s^{b_2}, p), B} := w^{B_2} : \texttt{unit}^{b_2} \quad (22)$$

$$M, l : t_1 \vdash \mu, (l \mapsto \{t_1'\}^q v) \quad (23)$$

By (22) and lemma 6:

$$pc; \text{-}; M, l : t_1 \vdash l^{(s^{b_2}, p), B} : \texttt{ref}^{\, b} \; s^{b_2} \text{ where } s^{b_2} \sim t_1 \text{ and } B \sqsubseteq b \text{ and } b \sqsubseteq b_2 \text{ and } pc \sqsubseteq b_2 \quad (24)$$

$$pc; \text{-}; M, l : t_1 \vdash w^{B_2 \sqcup PC \sqcup B} : s_1^{B_2 \sqcup PC \sqcup B} \text{ where } s_1^{B_2 \sqcup PC \sqcup B} \prec s^{b_2 \sqcup PC \sqcup B} \quad (25)$$

By (23), (24), and (25):

$$M, l : t_1 \vdash \mu, (l \mapsto \{s^{b_2 \sqcup PC \sqcup B}\}^{pq} w^{B_2 \sqcup PC \sqcup B}) \quad (26)$$

which is the desired heap typing result. The expression typing follows immediately by subtyping and $pc \sqsubseteq b_2$ (24).

6) Cases R-Ctxt, R-Protect-Ctxt: The result follows by lemma 7, lemma 5 and the induction assumption.

7) Cases R-Cast-$*$: The result follows straightforwardly with lemma 8.

8) Cases R-GuardCast-$*$: Using lemma 4 in case *R-GuardCast-Protect* and lemma 6 in case *R-GuardCast-Value*, the result follows straightforwardly.

APPENDIX B.
PROOF OF PROGRESS

If $pc; \text{-}; M \vdash e : t$ and $M \vdash \mu$ then, for all $PC \sqsubseteq pc$ either (i) e is a value, (ii) there exists p such that $e \,/\, \mu \xrightarrow{PC} \Uparrow p$, or (iii) there exist μ' and e' such that $e \,/\, \mu \xrightarrow{PC} e' \,/\, \mu'$.

Proof by induction on the derivation of $pc; \text{-}; M \vdash e : t$. The cases *T-Var*, *T-Int*, *T-Unit*, and *T-Addr* are trivial.

T-Protect
Case: e is not a value
By the induction assumption, either rule *R-Ctxt-Protect* or rule *R-Ctxt-Protect-Fail* applies.
Case: e is a value
Rule *R-Protect* applies.

T-App
Case: e_1 or e_2 is not a value
By the induction assumption, either rule *R-Ctxt* or rule *R-Ctxt-Fail* applies.
Case: e_1 and e_2 are values
Rule *R-App* applies.

The cases for ***T-New***,***T-GuardCast-$*$***, ***T-FunCast-$*$***, and ***T-RefCast*** work similarly as case *T-App*.

T-Sub
The result follows by the induction assumption.

T-Deref
Case: e is not a value
By the induction assumption, either rule *R-Ctxt* or rule *R-Ctxt-Fail* applies. Case: e is a value The typing assumption yields

$$\mu = \mu', (l \mapsto \{t'\}^p v) \quad (27)$$

With this result, rule *R-Deref* applies.

T-Asgn
Case: e_1 or e_2 is not a value
By the induction assumption, either rule *R-Ctxt* or rule *R-Ctxt-Fail* applies.
Case: e_1 and e_2 are values The typing assumption yields

$$\mu = \mu', (l \mapsto \{t'\}^p w^B) \quad (28)$$

$$pc; \text{-}; M \vdash l^{(t,q), B_1} := v : \texttt{unit}^b \quad (29)$$

- Case $PC \sqcup B_1 \sqsubseteq B$: *R-Asgn* applies.
- Case $PC \sqcup B_1 \not\sqsubseteq B$: *R-Asgn-NSU-Fail* applies.

T-Cast
Case: e is not a value
By the induction assumption, either rule *R-Ctxt* or rule *R-Ctxt-Fail* applies. Case: e is a value w^B with type s^b
The compatibility requirement of the typing assumptions yields

$$w \xrightarrow{s1 \Leftarrow s}_p w_1 \quad (30)$$

where $s_1^{b_1}$ is the destination type of the cast.

- Case $B \not\sqsubseteq b_1$: Rule *R-Cast-Sub-Fail* applies.
- Case $b_1 = ?$: Rule *R-Cast-To-Dyn* applies.
- Case $B \sqsubseteq b_1$, $b_1 = B_1$: Rule *R-Cast-Sub* applies.