Mechanized Type Safety for Gradual Information Flow

Tianyu Chen  
Computer Science  
Indiana University  
Bloomington, USA  
chen512@iu.edu

Jeremy G. Siek  
Computer Science  
Indiana University  
Bloomington, USA  
jsiek@indiana.edu

Abstract—We model a security-typed language with gradual information flow labels in a proof assistant, demonstrate its potential application to parsing and securing sensitive user input data, present the semantics as a definitional interpreter, and prove type safety. We compare the language features and properties of various existing gradual security-typed languages, shedding light on future designs.

Index Terms—gradual typing, information flow security, mechanized metatheory

I. INTRODUCTION

In this paper, we prove type safety for a language with gradual information flow labels. That is, a language in which the programmer can request that information flow be checked statically or they can defer such checking until runtime by using the unknown information flow label, written $\hat{i}$, in type annotations. We describe how information-flow secure language could be applied to constructing a parser that protects sensitive user input. Specifically, we focus on GLIO, a language introduced by de Amorim et al. [1], who prove that it satisfies noninterference [1] section 5) and the gradual guarantees [1] section 6). However, there are several more properties that one expects of such a language: type safety, blame safety, conservativity, and dynamic embedding [2]. In this paper we focus on the first of those properties, type safety. Unfortunately, the denotational semantics of GLIO given by de Amorim et al. [1] does not distinguish between trapped and untrapped errors, which makes it impossible to formulate the type safety property. To remedy this we present a definitional interpreter for GLIO that distinguishes between two kinds of trapped errors (failure of a no-sensitive-upgrade (NSU) check or a cast from high to low security) and the other errors which are untrapped. Our type safety proof, as usual, guarantees that untrapped errors never occur. The definitional interpreter and the proof of type safety are mechanized in the Agda proof assistant.

Broadly speaking, we are interested in confidentiality, that is, restricting information access to authorized parties only, which forms a triad together with integrity and availability as the foundation of information security [3]. From the perspective of a programming language, confidentiality is often formalized as satisfying noninterference, a theorem stating that high-security input must not affect publicly observable low-security outputs [4].

Modern software applications often accept user input where selected fields are sensitive, whose confidentiality is required during both parsing and processing. Suppose we have a web application that receives three fields from its user: 1) first name 2) last name 3) social security number, the grammar of which is defined in figure [1] where terminals are divided into low-security and high-security. The digits $d$ for social security number, being confidential to users of the web application, are of high-security, so they are marked red, while other terminals, such as the keys of the record and the strings $w$ for first name / last name, being safe to disclose, are all of low-security, marked in blue.

Consider the following user input: 
{FirstName=Mad;LastName=Hatter;SSN=012-34-5678}

The author of the web application could implement a parser for the grammar in Figure [1] in a language that enforces information flow security. Each terminal in the grammar would be labeled with a security level and the language would guarantee that the high-security information is only present in those parts of the output parse tree that are marked as high-security. For example, according to the grammar in figure [1] the example user input string is parsed into the parse tree in figure [2] where the terminal nodes that represent digits of the social security number are of high-security, while the terminals that compose the rest of the input string are of low-security. The confidentiality of SSN is guaranteed during data processing since the language enforces noninterference. When the web application interacts with the outside world, such as making a foreign function interface (FFI) call or storing into

\[
\langle RECORD \rangle ::=\langle FirstName=(ID); \rangle \\
\langle LastName=(ID); \rangle \\
\langle SSN=(SSN) \rangle \\
\langle ID \rangle ::=w, w \in \{A, ... Z, a, ... z\}^+ \\
\langle SSN \rangle ::=\langle D \rangle\langle D \rangle\langle D \rangle\langle D \rangle\langle D \rangle\langle D \rangle\langle D \rangle\langle D \rangle\langle D \rangle \\
\langle D \rangle ::=d, d \in \{0, ... 9\}
\]

Fig. 1: Example grammar for user input

\[
\{FirstName=Mad;LastName=Hatter;SSN=012-34-5678\}
\]
a database, the conceptual language need to encrypt whatever values labeled as high security before they are passed into a foreign routine.

The interest in enforcing confidentiality and regulating the flow of information in a computer program arises with its defense applications in the 1970s [5]. Denning [6] builds a information flow model using a lattice of security labels and Denning and Denning [7] discuss the certification technique in further detail, with a proof that a certified program will not give away confidential input from non-confidential output. Volpano et al. [8] propose a typed-based approach to enforcing information flow, by defining a type system for an imperative programming language and proving its security with a type soundness proof. This idea is further developed by Zdancewic and Myers [9]. Such a protection scales well since type checking is compositional [10]. There are projects that apply similar techniques but to other languages such as bytecode intermediate languages [11], object-oriented languages [12], and reactive programming languages [13]. Although the aforementioned languages are mostly theoretical, efforts have also been made to integrate information flow control into widely-used existing languages such as Jif for Java [14] and Flow Caml for OCaml [15] [16].

While type systems rule out undesired flows statically, it is also possible to do so at runtime. Li and Zdancewic [17] add flow control to Haskell by utilizing existing language features, specifically arrow and typeclass, to implement checks. Similarly, the LIO library employs a labeled IO monad to keep track of the current privilege level, which restricts both the observability and the security effect [18] [19]. LIO also provides 1) first-class labels so that labels are values and can be manipulated by the programmer on-the-fly, and 2) coarse-grained labeling so that a programmer may choose to label a value when it is necessary to impose flow control policies and omit the labels for security-insensitive parts of the program.

The HLIO [20] library introduces hybrid checking of information flow, whereby a programmer can choose between static or dynamic checking in different parts of the program. By default the checking is static, but a programmer can insert a defer clause to say that the security constraints should be checked at runtime.

Gradual typing [21] [22] [23] is a paradigm that combines static typing and dynamic typing - checks are performed at the boundaries between statically and dynamically typed code fragments. The most obvious difference from static typing is that a gradual type system usually contains a dynamic type that stands for the type that is statically unknown - in our case, it is a dynamic information flow label written $\mathcal{L}$. Unlike the hybrid approach discussed above, the programmer does not insert casts or defer clauses to mark the transitions between static and dynamic.

Recently there has been increasing interest in building gradual security-typed languages. The benefit of having a gradual security-typed language is that the programmer can choose when it is appropriate to put in the effort to pass the static security checks and when it is appropriate to defer such enforcement to runtime - where information flow violations will appear as trapped errors. Gradual typing facilitates migration between the static and dynamic checking because, roughly speaking, changes in type annotations are guaranteed to preserve program behavior, a property called the gradual guarantee [24].

GLIO [11], on which this paper is based, is comparable to HLIO in that both enable deferring information flow checks to runtime and have similar features to the LIO language mentioned above. They are different in that the latter is gradual instead of hybrid, checks are guided by type and a developer does not need to embed explicit casts into the program. Additionally, GLIO satisfies criteria of a gradually-typed language, such as dynamic and static gradual guarantees, making the migration between paradigms easy. Apart from GLIO, there are a few other noteworthy designs: Disney and Flanagan [24] explore the idea of adding explicit casts to enable dynamic flow control for a purely functional security-typed language called $\lambda_{gif}$. Fennell and Thiemann [25] present a similar cast calculus, ML-GS, with mutable references, a feature that $\lambda_{gif}$ lacks. The heap of ML-GS stores both values and types, a model that GLIO follows. Toro et al. [26] derive a language, GSLRef, leveraging the AGT framework [27]. GSLRef contains both a surface language and a cast calculus. NSU checks are derived from their static counterparts during the cast-insertion (called elaboration in GSLRef) procedure. The authors also show that the typing rules of GSLRef stay relatively similar to its fully static version SSLRef, other than a few relations being replaced by their gradual counterparts.

With the growth in gradual security-typed language designs,
we find it beneficial to make a thorough comparison between them. We are interested in their language features and properties. Additionally, their heap models are also worth studying, since a few of the designs store additional information beside a value at each memory cell.

To summarize, the contributions of this work are:

- A definitional interpreter for GLIO that distinguishes between different types of trapped errors.
- We prove type safety using the aforementioned semantics: a well-typed GLIO program never gets stuck and always evaluates to a well-typed machine configuration.
- We compare various existing gradual security-typed programming languages and discuss their design choices. We provide insight into which criteria future designs should satisfy.

II. REVIEW OF THE GLIO LANGUAGE

In this section, we briefly review the syntax of GLIO and the type system in the form of Agda code. GLIO is a dependently-typed programming language and proof assistant. The code and machine-checked proofs in this paper are available at https://github.com/Gradual-Typing/lambda-sec/tree/master/glio.

A. GLIO by Example

GLIO is a gradually-typed language. Gaps between the statically-typed fragments and the dynamically-typed fragments are bridged by implicit casts, which also serve as runtime information flow checks. Consider the program in listing that demonstrates how a function $g$, whose parameter has a dynamic information flow label, interacts with a statically typed region of code. Suppose there are two security levels: Low and High. The type annotation on each let binding is omitted and defaults to the type of the expression on the right-hand side. The program counter annotation of each $\lambda$-abstraction defaults to Low. A value with type $T$ protected by a certain information flow label $\ell$ inhabits $\text{Lab } \ell \ T$; the label may optionally be determined at runtime, written $\text{Lab } \ell \ T$, where the $\ell$ means a statically unknown label.

```
let f = \ x : (Lab Low Bool) . display x in
let g = \ x : (Lab \ell Bool) . (f x) in
let v = to-label High true in
f v
```

Listing 1: Example of casts

Listing 1 shows a well-typed program. However, it may leak information, since function $f$ publishes a low-security variable, while the value passed through function $g$, whose parameter is of a statically unknown security level, is a high-security boolean. To ensure security, GLIO terminates the execution due to a failed cast and prevents the high-security value from being disclosed. At the application $g \ v$, the boolean labeled High that variable $v$ is bound to is cast from Lab High Bool to Lab $\ell$ Bool, which is permitted. The application $f \ x$ in the body of $g$, on the other hand, attempts to cast a high-security boolean from Lab $\ell$ Bool to Lab Low Bool, which errors due to a failed runtime check.

If every annotation is decorated with a concrete label, GLIO can discover illegal information flows during type checking, just like a statically security-typed language. Consider the following program, Low and High:

```
let f = \ x : (Lab Low Bool) . x in
let v = to-label High true in
f v
```

Listing 2: Example of a statically-typed program, rejected

This program is unsafe since a high-security boolean is passed into the identity function $f$ which takes a low-security boolean, ruled out by the type system. To make the program type check, the programmer may lower the label on $v$ like in a statically security-typed language:

```
let f = \ x : (Lab Low Bool) . x in
let v = to-label Low true in
f v
```

Listing 3: Example of a statically-typed program, fixed

GLIO is a coarse-grained security-typed language, which means that not all types and values are labeled. Values of unlabeled types default to publicly visible. It also provides first-class labels (labels as values). Consider the following variation of the examples in listings and :

```
let \ell = (user-input) in
let f = \ x : (Lab Low Bool) . x in
let v = to-label-dyn \ell true in
f v
```

Listing 4: Example of dynamically labeled value

There are two changes in the example above compared with its static variants: 1) The to-label clause is replaced with to-label-dyn. 2) A variable $\ell$, bound to the user’s input, is provided instead of a concrete label. If the user input label is High when the program executes, a check happens during the application, casting a boolean value labeled as high-security to the type Lab Low Bool, which fails and triggers a cast error, thus preventing the program from leaking information. If the user input is Low instead, the program finishes successfully and returns a boolean of low-security.

Similarly, consider the following example, where a new heap location is created to store a secret:

```
let x = to-label High true in
let y = unlabel x in
new Low y
```

Listing 5: Creating labeled reference, statically rejected

The boolean is labeled as High, unlabeled and then written to a new heap location when security level Low. It contains an unsafe information flow from high-security to low-security and is ruled out by the type checker. We may assign the secrecy of the newly created heap cell at runtime:
If the input label \( \ell \) is High when the program runs, it finishes successfully since the information is flowing from high-security to high-security; otherwise if \( \ell \) is Low, an NSU error occurs due to a failed runtime check, preventing the program from leaking information. We explain different types of errors in further detail in section \ref{III}.

B. Defining the Syntax in Agda

The information flow label is defined as a datatype \( \mathcal{L} \) with constructor \( \mathcal{L} \) that takes a natural number privilege level:

\[
\text{data } \mathcal{L} : \text{Set where}
\]
\[
\mathcal{L} : \mathbb{N} \to \mathcal{L}
\]

It forms a lattice with the order \( \preceq \), join \( \sqcup \), and meet \( \sqcap \), which are analogous to their natural number counterparts \( \leq \), \( \cup \), and \( \cap \). We use \( \check{\mathcal{L}} \) for the type of a gradual label - it can be either a concrete label or dynamic \( \hat{\ell} \):

\[
\text{data } \check{\mathcal{L}} : \text{Set where}
\]
\[
\check{\mathcal{L}} : \mathcal{L}^\land; \check{\mathcal{L}} : \mathcal{L} \to \check{\mathcal{L}}
\]

For Low and High, we have the following shorthand:

\[
\mathcal{L} = \mathcal{L}^0; \quad \mathcal{L}^\land = \mathcal{L}^0 \land \mathcal{L}; \quad \mathcal{H} = \mathcal{L}^1; \quad \mathcal{H}^\land = \mathcal{L}^0 \land \mathcal{H}
\]

The definitions of gradual join of labels (\( \land \)) and types (\( \land \)), gradual meet of labels (\( \land \)) and types (\( \land \)), and consistent subtyping of labels (\( \preceq \)) and types (\( \preceq \)) are all defined the same as in de Amorim et al. \cite{deAmorim} and thus omitted.

The types are defined using the gradual label datatype \( \check{\mathcal{L}} \):

\[
\text{data } \check{\mathcal{T}} : \text{Set where}
\]
\[
\check{\mathcal{T}} : \check{\mathcal{T}} \to \check{\mathcal{T}}; \quad \check{\mathcal{B}} : \check{\mathcal{T}}; \quad \check{\mathcal{L}} : \check{\mathcal{T}}
\]

\[
\text{Ref} : \check{\mathcal{L}}^\land \to \check{\mathcal{T}} \to \check{\mathcal{T}}; \quad \text{Lab} : \check{\mathcal{L}}^\land \to \check{\mathcal{T}} \to \check{\mathcal{T}}; \quad \text{Function}
\]

Most cases are straight-forward. The label on the reference type denotes the secrecy level of the heap location. There are two labels decorated on a function type, which serve as the static program counters before and after the computation of the body of a \( \lambda \)-abstraction, corresponding to the two labels on the signature of the typing rule (introduced below).

The terms of the GLIO are defined in Agda using the abstract binding tree library (https://github.com/jsiek/abstract-binding-trees). The terms are extrinsically typed; a term \( M \) that is typed \( \mathcal{T} \) under typing context \( \Gamma \) and program counter labels \( \ell_1 \) and \( \ell_2 \) is written \( \Gamma \vdash \ell_1 \mathcal{G} \ell_2 : \mathcal{T} \). The typing context is defined as a list of types (\( \text{List } \mathcal{T} \)), since we use De Bruijn notation for variables. The two program counter labels \( \ell_1 \) and \( \ell_2 \) are for the security effects before and after the computation, which restrict heap operations in their respective scopes. We omit the complete definitions of typing rules since they are ported from the GLIO paper \cite{deAmorim}; A-normal form is used so that the syntax stays close.
C. The Definitional Interpreter of GLIO

The interpreter is defined as a total function \( \mathcal{V} \) that depends on two helpers, \( \text{castL} \) and \( \text{castT} \), whose definitions are detailed in appendix [A]. There are cases where \( \text{castT} \) could possibly get stuck. We show that those cases are never encountered by proving type safety in section [IV].

For simplicity, we show only the interesting cases of \( \mathcal{V} \), which are split into three parts: abstraction and application (figure [3]), heap access (figure [4]), and labeling operations (figure [5]). The interpreter \( \mathcal{V} \) takes an environment \( \gamma \), a well-typed term \( M \), an original store \( \mu \), a program counter \( pc \), and a natural number \( k \) called gas which makes \( \mathcal{V} \) total.

If gas runs out, \( \mathcal{V} \) returns a timeout, shown in the first case of figure [3]. Variable lookup is straightforward, which simply returns the value that variable \( x \) corresponds to in the environment \( \gamma \). If the lookup fails, it means that the term is open and we get stuck. When evaluating if, we dispatch on the value of the condition \( x \). We go to the first branch if \( x \) is \( \mathcal{V} \)-true, the second if it is \( \mathcal{V} \)-false, and get stuck if it is neither. In either case, the sub-term is evaluated, the label is cast to be the gradual join of the two branches, and the value from the sub-term is then cast to the gradual join of types from both branches.

In \( \lambda \)-abstraction’s case, we build a closure by wrapping the well-typed body \( N \) and the current environment \( \gamma \) in \( \mathcal{V} \)-clos. Function application is defined leveraging an auxiliary helper apply that applies a value which is either a \( \mathcal{V} \)-clos or a \( \mathcal{V} \)-proxy and otherwise gets stuck. If the value is a closure, we directly dive into the body \( N \) with an extended environment \( w : \rho \), where \( \rho \) is the environment wrapped in the closure. On
\[ \forall \Gamma T \ell_1 \ell_2 . (\gamma : \text{Env}) \rightarrow (M : \text{Term}) \rightarrow (pc : \mathcal{L}) \rightarrow (k : \mathbb{N}) \rightarrow \text{Result Conf} \]

\[ \forall \gamma (\text{get } (\ell x)) \mu pc (k + 1) = \begin{cases} 
\text{cast} T \mu (pc \sqcup \ell_2) T' T v, & \text{if } \mu[[n, \ell_1, \ell_2]] \equiv (T', v) \\
\text{error stuck} & \text{if } \mu[[n, \ell_1, \ell_2]] \text{ is undefined} \\
, \text{if } \gamma[x] \equiv \text{Vref } \langle n, \ell_1, \ell_2 \rangle \\
\text{error stuck} & \text{otherwise} 
\end{cases} \]

, where \( \Gamma[x] \equiv \text{Ref } \ell T \)

\[ \forall \gamma (\text{set } (\ell x) (\ell y)) \mu pc (k + 1) = \begin{cases} 
\text{do} \\
(\mu', v', pc') \leftarrow \text{cast} T \mu pc T' T v \\
(\mu'', v'', pc'') \leftarrow \text{cast} T \mu' pc' T T'' v' \\
\text{setmem} \mu'' (n, \ell_1, \ell_2) pc'' (T'', v'') & \text{if } \mu[[n, \ell_1, \ell_2]] \equiv (T'', v'') \\
\text{error stuck} & \text{if } \mu[[n, \ell_1, \ell_2]] \text{ is undefined} \\
, \text{if } \gamma[x] \equiv \text{Vref } (n, \ell_1, \ell_2) \text{ and } \gamma[y] \equiv v \\
\text{error stuck} & \text{otherwise} 
\end{cases} \]

, where \( \Gamma[x] \equiv \text{Ref } \ell T, \Gamma[y] \equiv T'' \)

\[ \forall \gamma (\text{new } \ell (\ell y)) \mu pc (k + 1) = \begin{cases} 
\text{result } \langle (n, pc, \ell) \rightarrow (T, v) \equiv \mu, \text{Vref } (n, pc, \ell, pc) \rangle & \text{, where } n \text{ is fresh, if } \gamma[y] \equiv v \\
\text{error stuck} & \text{otherwise} \\
\text{error NSUError} & \text{otherwise} 
\end{cases} \]

, where \( \Gamma[y] \equiv T \)

\[ \forall \gamma (\text{new}_{\text{dyn}} (\ell x) (\ell y)) \mu pc (k + 1) = \begin{cases} 
\text{result } \langle (n, pc, \ell) \rightarrow (T, v) \equiv \mu, \text{Vref } (n, pc, \ell, pc) \rangle & \text{, where } n \text{ is fresh, if } pc \sqsubset \ell \\
\text{error NSUError} & \text{otherwise} \\
, \text{if } \gamma[x] \equiv \text{Vlabel } \ell \text{ and } \gamma[y] \equiv v \\
\text{error stuck} & \text{otherwise} 
\end{cases} \]

, where \( \Gamma[y] \equiv T \)

\[ \text{setmem} : (\mu : \text{Store}) \rightarrow \text{Location} \rightarrow (pc : \mathcal{L}) \rightarrow T \times \text{Value} \rightarrow \text{Result Conf} \]

\[ \text{setmem} \mu (n, \ell_1, \ell_2) pc tv = \begin{cases} 
\text{result } \langle (n, \ell_1, \ell_2) \rightarrow tv \equiv \mu, \text{Vset } pc \rangle & \text{, if } pc \sqsubset \ell_2 \\
\text{error NSUError} & \text{otherwise} 
\end{cases} \]

Fig. 4: The definitional interpreter of GLIO: heap memory operations

The other hand if the value is a function proxy, we first cast the type of the domain and the program counter at the beginning of the computation, after which we recursively call apply. Then we cast the program counter after the computation and the type of the codomain, on the value after the application. In short, applying a proxy unwraps one layer of V-proxy into casts. The application case of \( \forall \) casts the domain on the value \( w \) and the program counter and subsequently calls apply.

The semantics for heap operations are shown in figure 2. When reading from a heap location \( \langle n, \ell_1, \ell_2 \rangle \text{ typed Ref } \ell T \), we first look it up in \( \mu \). If the index is out-of-bound, we run into a memory access error and the evaluation gets stuck. Otherwise if the type-value pair on heap \( \mu \) is \( (T', v) \), we cast the value \( v \) from \( T' \) to \( T \), since we expect a \( T \) from the heap reference. When writing to a heap location, we use an auxiliary function setmem to check for NSU error, since writing to a location that is less secure than the current program counter must be strictly ruled out. If the index is out-of-bound, we get stuck. Otherwise, we cast the value to store from its original type \( T' \) to the type on the reference, \( T \), and then to the type annotation on the heap cell, \( T'' \). Finally we invoke setmem to write the value to the heap. The operation new and new-dyn create a new cell on the heap. The difference is whether the label of secrecy \( \ell \) comes statically from the term or dynamically from the environment \( \gamma \).

Figure 5 shows the cases for unlabel and to-label. When unlabeling a value, apart from peeling off the label \( \ell \) and retrieving the unwrapped value \( v \), we need to upgrade the program counter by joining with \( \ell \) - this is why the example program in listing 6 may fail at runtime. The cases for to-label and to-label-dyn are the same except that in the former the label \( \ell \) comes from the term while in the latter \( \ell \) comes from the environment \( \gamma \) - similar to the difference between new and new-dyn. The two labeling operations both evaluate the sub-term \( M \), perform an NSU check, and return the value \( v \) labeled with \( \ell \).
Well-typed heap:

\[ \forall \gamma \, \forall T \, \ell_1, \ell_2 \cdot (\gamma : \text{Env}) \rightarrow (M : \text{Term}) \rightarrow (\mu : \text{Store}) \rightarrow (pc : \mathcal{L}) \rightarrow (k : \mathbb{N}) \rightarrow \text{Result Conf} \]

\[
V \; \gamma \; (\text{unlabel} \; (\cdot \; x)) \; \mu \; \text{pc} \; (k + 1) = \begin{cases} 
\text{result } \langle \mu, v, \text{pc} \sqcup \ell \rangle, & \text{if } \gamma[x] \equiv V_{\text{lab}} \; \ell \; v \\
\text{error stuck}, & \text{otherwise}
\end{cases}
\]

\[
V \; \gamma \; (\text{tolabel} \; \ell \; M) \; \mu \; \text{pc} \; (k + 1) = \begin{cases} 
\text{result } \langle \mu', V_{\text{lab}} \; \ell \; v, \text{pc} \rangle, & \text{if } \text{pc}' \not\equiv \text{pc} \sqcup \ell \\
\text{error } \text{NSUError}, & \text{otherwise}
\end{cases}
\]

\[
V \; \gamma \; (\text{tolabel}_{\text{dyn}} \; (\cdot \; x) \; M) \; \mu \; \text{pc} \; (k + 1) = \begin{cases} 
\text{result } \langle \mu', V_{\text{lab}} \; \ell \; v, \text{pc} \rangle, & \text{if } \text{pc}' \not\equiv \text{pc} \sqcup \ell \\
\text{error } \text{NSUError}, & \text{otherwise}
\end{cases}
\]

Fig. 5: The definitional interpreter of GLIO: labeling

Fig. 6: Well-typed environment, heap, and computation result

Well-typed environment:

\[
\frac{\mu \vdash v : T}{\langle \cdot \rangle ; \mu \vdash \cdot : \Gamma ; \mu \vdash \gamma}
\]

Well-typed heap:

\[
\frac{\mu \vdash \cdot : \Gamma ; \mu \vdash \gamma}{\mu \vdash \cdot : \Gamma ; \mu \vdash \cdot : \gamma}
\]

Well-typed computation result:

\[
\frac{\mu \vdash \mu', \mu \vdash v : T}{\vdash \text{result } \langle \mu, v, \text{pc} \rangle : T}
\]

\[
\frac{\mu \vdash \mu, \mu \vdash v : T}{\vdash \text{timeout} : T}
\]

\[
\frac{\mu \vdash \mu, \mu \vdash v : T}{\mu \vdash \text{castError} : T}
\]

\[
\frac{\mu \vdash \mu, \mu \vdash v : T}{\mu \vdash \text{NSUError} : T}
\]

We can see that our interpreter follows a similar structure to the denotational semantics in de Amorim et al. [1] figure. 13, except than the authors use CPO sets to represent the denotation of terms and casts, while we present the semantics as an interpreter that employs a machine configuration monad, which is easier to reason about in a proof assistant, since the mechanized proof simply follows the branches in the interpreter’s code. Another benefit is that we can view the evaluation in action - in fact, all the examples shown in section II are runnable in this interpreter.

IV. MECHANIZED TYPE SAFETY PROOF IN AGDA

We are now ready to prove type safety with the operational semantics in section II. It is proved by showing the computation result of the interpreter is always well-typed, given well-typed input.

The definitions of well-typedness for environment, heap, computation, and value are given in figure 6 and 7. The typing of environment is quantified by a typing context \( \Gamma' \) and a store typing \( \mu \). We use the heap itself as the store typing context, as we store the type of each cell together with the value. A well-typed environment means that each value in it is well-typed according to its corresponding type in \( \Gamma' \). The typing of value is quantified by the store typing \( \mu \), which is necessary due to mutable reference. In the two cases of reference, we only require that the heap location is valid; the type in the cell may not necessarily be the same as the one on the reference. The typing of heap is straightforward; since we store types directly to mutable reference. In the two cases of reference, we only require that the heap location is valid; the type in the cell may not necessarily be the same as the one on the reference. The typing of heap is straightforward; since we store types directly on heap, the type and value need to jive in each cell. Finally, every result may be well-typed except stuck - we would like to prove that the program never gets stuck. For a configuration \( \langle \mu, v, \text{pc} \rangle \) to be well-typed, both the heap \( \mu \) and the value \( v \) must be well-typed.

The statement of type safety is in theorem 2 which is a corollary of proposition 1.

**Proposition 1** (The interpreter \( V \) is type safe). If the initial heap \( \mu \) is well-typed \( \mu \vdash \mu \), the initial environment is well-typed \( \Gamma ; \mu \vdash \gamma \), and the term is well-typed \( \cdot \vdash \ell_1, \ell_2 \; M : T \), then the evaluation result is well-typed \( \Gamma ; \mu \vdash V \; \gamma \; \mu \; \text{pc} \; k : T \).

The detailed proof is discussed in appendix B.

**Theorem 2** (Type safety). If term \( M \) is well-typed:

\[
\cdot \vdash \ell_1, \ell_2 \; M : T
\]

then evaluating \( M \) gets a well-typed result:

\[
\vdash \cdot \vdash \cdot \; M \; \text{pc} \; k : T
\]

**Proof.** This theorem is a special case of proposition 1 where the initial environment and heap are both empty, which are trivially well-typed. \( \blacksquare \)
V. COMPARING GRADUAL SECURITY-TYPED LANGUAGE DESIGNS

In this section we compare four noteworthy designs - $\lambda_{gf}$ [24], ML-GS [25], GSLRef [26], and GLIO [1].

A. Language Features

Table I compares the features of the four languages, which fall into three categories: the design of the language (whether it provide implicit or explicit casts, which is the distinction between a gradually typed surface language versus a cast calculus that is meant to serve as an intermediate language), the heap model (how the language handles mutable references), and the labeling granularity (in what places information flow labels may appear).

a) Language design: $\lambda_{gf}$ and ML-GS provide explicit casts but no implicit ones, so they are cast calculi. GSLRef provides implicit casts; it is a gradually-typed language that is derived from its statically-typed sister language SSLRef. GLIO also provides implicit casts as we discussed in section II. It does not have a statically-typed sister language.

b) Heap model: $\lambda_{gf}$ does not have mutable reference. Both ML-GS and GLIO choose to store a value together with its type on heap and generate a cast from the heap type to the type of the reference at runtime. GSLRef, on the other hand, stores casts represented as evidence on the heap. Although the approach of GLIO sounds similar to the one of ML-GS, there is a type invariant of GLIO that ML-GS lacks. In GLIO the type of a cell stays the same across updates, which is enabled by reading the type from the address first, followed by casting the value into that type. As shown in figure 4, the $T''$ stays unchanged in the set case. The heap model of ML-GS does not enforce this invariant; the $R\rightarrow$Asgn rule makes it possible to completely replace the raw type of a cell, so the cast may fail when reading from a reference.

These heap models that insert casts at runtime when reading and writing create a challenge for assigning blame. GLIO does not perform blame tracking. On the other hand, ML-GS does perform blame tracking and propagates blame labels through the heap to assign blame when dereferencing. However, this creates a problem regarding the statement of the blame theorem, which usually says that if each cast labeled $p$ in term $M$ of the cast calculus is a safe cast, then $M$ will not reduce to blame $p$. But the types stored on the heap are only known when a program executes. Although the authors of ML-GS conjecture that “an extension” of the blame theorem could hold, they do not provide a theorem statement or a proof.

c) Labeling granularity: GLIO has first-class labels since it is based on the LIO library, where labels are treated as values. GLIO also follows LIO and HLIO and employs coarse-grained labeling, in which not all values are labeled by default and a programmer need to use $\text{to\text{-}label}$ to explicitly protect a value. However, it is proved that coarse-grained labeling is equally expressive as fine-grained labeling, where every value is labeled [30].

d) Language Feature Summary: Overall GLIO and GSLRef are the more feature-rich languages among the four. GLIO supports coarse-grained labeling and first-class labels, which makes it easier to migrate from legacy code that does not have information flow labels. However, GLIO does not perform blame tracking, so it can’t satisfy a blame theorem. Adding blame tracking it challenging because of the heap model, though perhaps the ideas of Siek et al. [31] may be applicable. GSLRef offers insight into deriving the gradual security-typed language from its statically typed sister language. Although it lacks first-class labels, its fine-grained labeling scheme means a label comes with each value.

B. Theorems and Properties

Table II summarizes the metatheoretic properties that the four languages satisfy. The “maybe” option is for properties that are either conjectured by the authors or that we suspect they would satisfy. Noninterference and type safety are satisfied by all of the languages. We discuss the other three properties in further detail:

a) Gradual guarantees: The static gradual guarantee states lowering the type precision of a term does not introduce a static type error. The dynamic gradual guarantee states that lowering the type precision of a term does not change its runtime behavior. On the other hand, increasing the type precision of a term may trigger a cast error (e.g. by adding an incorrect type annotation) but otherwise the behavior remains unchanged. Neither $\lambda_{gf}$ nor ML-GS discuss the gradual guarantees, as they preceeded the invention of the gradual guarantees. GSLRef satisfies the static gradual guarantee but
not the dynamic gradual guarantee, which the authors claim is in tension with noninterference. GLIO resolves this tension by having casts check labels only, without classifying the data.

b) **Blame theorem**: The blame theorem says that if every cast labeled by \( p \) in a term \( M \) is safe by satisfying a subtyping relation, then \( M \) will not reduce to a blame \( p \). The authors of \( \lambda_{gif} \) proved the blame theorem but \( \lambda_{gif} \) lacks mutable references. The authors of \( ML-GS \) conjecture that the language satisfies a blame theorem but do not state the theorem or give a proof. Neither \( GSL_{Ref} \) nor GLIO performs blame tracking.

c) **Space efficiency**: The operational semantics of \( \lambda_{gif} \), \( ML-GS \), and GLIO both use function proxies that can build up, so they are not space efficient. \( GSL_{Ref} \) utilizes the AGT framework which can in principle enable space efficiency \([32, 33]\), but space efficiency is not discussed in the \( GSL_{Ref} \) paper.

### VI. Conclusion and Future Work

In this paper we briefly reviewed the design of a gradual security-typed language, GLIO, defined its semantics with a definitional interpreter and provided a mechanized proof of type safety in Agda. Based on our comparison and analysis of four existing language designs, we recommend that a gradual security-typed language have the following characteristics.

- A **gradual language and a cast calculus**. The language design should include both a gradual surface language with implicit casts and a cast calculus with explicit casts.
- **Information flow control with fine-grained labeling**. Since fine-grained labeling and coarse-grained labeling are equally expressive, we choose fine-grained labeling, where each value is labeled and (gradual) labels are embedded into the types, thus providing a more uniform syntax. Different from \( GSL_{Ref} \) but similar to GLIO, values should have a concrete label. An unlabeled value should be shorthand for a default low-security label. We conjecture that this alleviates the problem with \( GSL_{Ref} \), where values can become dynamically typed and lose their original label when changing type annotations to be less precise.
- **Embedding of static and dynamic information flow control**. The gradual language should include both static and dynamic information flow control. This means that the NSU checks should be expressed as casts, similar to the path that \( GSL_{Ref} \) follows. We are able to formally define the static and dynamic extremes leveraging this approach.
- **Mutable reference and proxies**. The language should support mutable reference. However, unlike \( ML-GS \) and GLIO, the standard approach involving reference proxies and a simple heap (that maps addresses to values) should be investigated \([34]\).
- **Blame tracking**. The language should support blame tracking.
- **Space efficiency**. The language should be space efficient.

We plan to investigate a language with these characteristics and develop its metatheory. Hopefully it will meet all the criteria discussed in section 5.3.

### References

APPENDIX A
DETAILED DEFINITIONS OF CASTS BETWEEN LABELS AND TYPES

Figure 8 shows the definitions of the cast functions. \( V_{\text{close}} \) \( M \gamma \) denotes a closure value with a well-typed body \( M \) and an environment \( \gamma \). \( V_{\text{proxy}} \quad \ell_1, \ell_2 \rightarrow T \ S' \quad \ell', \ell_2 \mapsto T' \ v \) stands for a function proxy - a value wrapped with two function types and a proof that the source is the consistent subtype of the target. \( \text{castL} \) guarantees the runtime program counter \( pc \) is the consistent subtype of the static one \( \ell_2 \). \( \text{castT} \) casts a value \( v \) from \( T_1 \) to \( T_2 \). It checks whether \( T_1 \) is the consistent subtype of \( T_2 \). There are some additional label checks in the \( \text{Ref} \ \ell \ T \) and \( \text{Lab} \ \ell \ T \) cases. A reference value \( V_{\text{ref}} \quad \langle n, \ell_1, \ell_2 \rangle \), whose secrecy is denoted by \( \ell_2 \), can inhabit either \( \text{Ref} \ \ell \ T \) or \( \text{Ref} \ \ell_2 \ T \) for some \( T \). Similarly, a labeled value \( V_{\text{lab}} \ \ell \ v \) can inhabit \( \text{Lab} \ \ell \ T \) for some \( T \) provided that \( \ell \not\approx \ell' \). If any of these checks fails, the cast results in a \( \text{castError} \). For example, if we attempt to make \( V_{\text{lab}} \ \text{Lab} \ \text{High} \ v \) to inhabit \( \text{Lab} \ \text{Low} \ T \) for some \( v, T \). When casting a value of function type, we use function proxy and wrap the source type, the target type, and a proof that the source is the consistent subtype of the target using \( V_{\text{proxy}} \). In figure 3 in the function application case of \( V \), we have showed how a function proxy is consumed.

APPENDIX B
DETAILED LEMMAS AND PROOFS OF SECTION IV

A. Supplementary Lemmas

**Lemma 3** (Variable lemma). If the environment is well-typed \( \Gamma; \mu \vdash \gamma \) and \( \Gamma[x] = T \), then:

\[ \exists v. \gamma[x] \equiv v \land \mu \vdash v : T \]

Proof. By induction on the De Bruijn index \( x \):

- If \( x = 0 \), the first value in a well-typed environment is always well-typed.
- If \( x = \text{suc} \ k \) for some \( k \), it is proved by the induction hypothesis about \( k \).

**Lemma 4** (Heap lookup). If the heap is well-typed \( \sigma \vdash \mu \) and \( \mu[\langle n, \ell_1, \ell_2 \rangle] \equiv \langle T, v \rangle \) for some location \( \langle n, \ell_1, \ell_2 \rangle \), then \( \sigma \vdash v : T \).

Proof. By induction on \( \sigma \vdash \mu \):

- If the heap is empty, this is impossible because there is no \( \langle T, v \rangle \) pair.
- If the \( \mu \) is not empty, there are two possibilities. If the location to lookup is the same as the index of the first cell, then \( v \) is typed at \( T \) since the heap is well-typed. If not, it is proved by the induction hypothesis about the rest of the heap.

**Lemma 5** (Cast \( \text{castT}' \) is well-typed). If \( T_1 \subseteq T_2 \), with a well-typed heap \( \mu \vdash \mu \) and a well-typed value \( \mu \vdash v : T_1 \), we have:

\[ \vdash \text{castT}' \mu \ pc \ T_1 \ T_2 \ v : T_2 \]

Proof sketch. By induction on the typing derivation of the value \( v \). Since \( v \) is well-typed, we never go into the stuck branches. The complete proof is mechanized in Agda.

**Lemma 6** (Updating preserves well-typed heap). If heap \( \sigma \) is well-typed \( \mu \vdash \sigma \), the address to update is valid \( \mu[\langle n, \ell_1, \ell_2 \rangle] \equiv \langle T, v \rangle \), the new value is well-typed \( \mu \vdash w : T \). Then:

\[ \langle n, \ell_1, \ell_2 \rangle \mapsto \langle T, w \rangle : \mu \vdash \sigma \]

Proof sketch. By induction on \( \mu \vdash \sigma \). The complete proof is mechanized in Agda.

**Lemma 7** (Creating new cell preserves well-typed heap). If heap \( \sigma \) is well-typed \( \mu \vdash \sigma \), the index \( n \) is fresh, the new value is well-typed \( \mu \vdash v : T \). Then:

\[ \langle n, \ell_1, \ell_2 \rangle \mapsto \langle T, v \rangle : \mu \vdash \sigma \]

Proof sketch. By induction on \( \mu \vdash \sigma \). The complete proof is mechanized in Agda.

B. Proving That the Interpreter Is Safe

The interesting cases in the proof of proposition 4 are detailed below.

**Proof sketch.** The complete proof is formalized in Agda. We only detail a few cases here. If the evaluation times out, it is trivially well-typed. Otherwise, by induction on the typing derivation of the term \( M \):

- **Case variable (\( \text{\texttt{\_\_}} \ x \)):** The environment \( \gamma \) is well-typed so this case is proved by lemma 3.

  **Case if:** By lemma 3 there are only two possibilities that \( x \) typed \( \text{\texttt{\_\_}} \) \( x \) may correspond to: \( \text{V-true} \) or \( \text{V-false} \), so the program does not get stuck during the lookup of \( x \). If \( x \) is \( \text{V-true} \) and we go to the then-branch \( M \), we can see the evaluation result of \( M \) - if \( M \) times out or errors by either \( \text{castError} \) or \( \text{NSUError} \) then the result is trivially well-typed; evaluating \( M \) does not get stuck because of the induction hypothesis. If it returns a machine configuration, we proceed with \( \text{castL} \), which either errors by a \( \text{castError} \), which is well-typed, or evaluates to a configuration. The final \( \text{castT} \) either errors by a \( \text{castError} \), which is again trivially well-typed, or invokes \( \text{castT}' \), the well-typedness of which is proved by lemma 5. The proof of the else-branch \( N \) follows the same structure.

  **Case get:** By lemma 3 the value that \( x \) with a reference type corresponds to must be of form \( \text{V-ref} \quad \langle n, \ell_1, \ell_2 \rangle \), whose typing may follow either the \( \text{Ref} \) rule or the \( \text{RefDyn} \) rule (in figure 7). In either case, we conclude that \( \mu[\langle n, \ell_1, \ell_2 \rangle] \equiv \langle T, v \rangle \), so the interpreter does not get stuck due to a failed heap lookup. Since the result of \( \text{castT} \) is well-typed (a corollary of lemma 5), the
\[\text{castL} : (\mu : \text{Store}) \rightarrow (pc : \mathcal{L}) \rightarrow (\ell_1, \ell_2 : \hat{L}) \rightarrow \text{Result Conf}\]

\[
\text{castL} \mu \ pc \ \hat{\ell}_1 \ \hat{\ell}_2 = \begin{cases} 
\text{result } \langle \mu, \mathcal{V}_\text{ret}, pc \rangle , & \text{if } pc \preceq \hat{\ell}_2 \\
\text{error } \text{castError} , & \text{otherwise}
\end{cases}
\]

\[\text{castT}' : (\mu : \text{Store}) \rightarrow (pc : \mathcal{L}) \rightarrow (T_1, T_2 : \mathbb{T}) \rightarrow (v : \text{Value}) \rightarrow \text{Result Conf}\]

\[
\text{castT}' \mu \ pc \ \hat{T}_1 \ \hat{T}_2 \rightarrow \begin{cases} 
\text{result } \langle \mu, \mathcal{V}_\text{ret}, pc \rangle , & \text{if } \hat{\ell}_2 \equiv \ell_2 \\
\text{error } \text{castError} , & \text{otherwise}
\end{cases}
\]

\[
\text{castT}' \mu \ pc \ \text{Ref} \ \hat{\ell}_1 \ \hat{T}_1 \ \text{Ref} \ \hat{\ell}_2 \ T_2 \rightarrow \begin{cases} 
\text{result } \langle \mu, \mathcal{V}_\text{ret}, \ell_1, \ell_2 \rangle , & \text{if } \hat{\ell}_2 \equiv \ell_2 \\
\text{error } \text{castError} , & \text{otherwise}
\end{cases}
\]

\[
\text{castT}' \mu \ pc \ \text{Lab} \ \hat{\ell}_1 \ T_1 \ \text{Lab} \ \hat{\ell}_2 \ T_2 \rightarrow \begin{cases} 
\text{result } \langle \mu', \mathcal{V}_\text{lab}, \ell' \rightarrow v, pc' \rangle , & \text{if } \ell' \preceq \hat{\ell}_2 \\
\text{error } \text{castError} , & \text{otherwise}
\end{cases}
\]

\[
\text{castT}' \mu \ pc \ S \ \hat{\ell}_1 \ \hat{\ell}_2 \ T \ \hat{\ell}_1 \ \hat{\ell}_2 \ T' \rightarrow v = \begin{cases} 
\text{result } \langle \mu, \mathcal{V}_\text{proxy}, \ell_1, \ell_2 \rangle , & \text{if } \mathcal{V}_\text{proxy} \leftrightarrow v, pc \\
\text{error } \text{castError} , & \text{otherwise}
\end{cases}
\]

Fig. 8: Casts between labels and types

The final result of \text{get} is well-typed (note that the heap is not modified by \text{castT} or \text{castL}).

\textbf{Case set:} The reasoning is similar to that of \text{get}. If the two casts both return valid configurations, we need to prove that the configuration after \text{setmem} is still well-typed. If the NSU check in \text{setmem} fails, an \text{NSUError} is well-typed. Otherwise, the well-typedness of the updated heap is justified by \text{lemma} [6].

\textbf{Case new-dyn:} Similar to \text{get} and \text{set}, we know that both variables correspond to well-typed values. If the NSU check fails, an \text{NSUError} is well-typed. Otherwise, the resulting configuration contains a heap extended with the new cell and a returned reference. The well-typedness of the extended heap is proved using [7]. The reference is well-typed due to the \text{RefDyn} rule.

\textbf{Case new:} Similar to new-dyn.

\[\square\]
## Appendix C

**Complete Definitions and Proofs in Agda**

<table>
<thead>
<tr>
<th>Definition or theorem</th>
<th>File</th>
<th>Agda function or datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label join (⊔)</td>
<td>StaticsGLIO.agda</td>
<td>Operator <em>⊔</em></td>
</tr>
<tr>
<td>Label meet (⊓)</td>
<td>StaticsGLIO.agda</td>
<td>Operator <em>⊓</em></td>
</tr>
<tr>
<td>Label partial order (≼)</td>
<td>StaticsGLIO.agda</td>
<td>Operator <em>≼</em></td>
</tr>
<tr>
<td>Gradual label join (⋎)</td>
<td>StaticsGLIO.agda</td>
<td>Operator <em>⋎</em></td>
</tr>
<tr>
<td>Gradual label meet (⋏)</td>
<td>StaticsGLIO.agda</td>
<td>Operator <em>⋏</em></td>
</tr>
<tr>
<td>Gradual label intersection (∏)</td>
<td>StaticsGLIO.agda</td>
<td>Operator <em>∏</em></td>
</tr>
<tr>
<td>Label consistent subtyping (≲)</td>
<td>StaticsGLIO.agda</td>
<td>Operator <em>≲</em></td>
</tr>
<tr>
<td>Type join (∨)</td>
<td>StaticsGLIO.agda</td>
<td>Operator <em>∨</em></td>
</tr>
<tr>
<td>Type meet (∧)</td>
<td>StaticsGLIO.agda</td>
<td>Operator <em>∧</em></td>
</tr>
<tr>
<td>Typing rules</td>
<td>StaticsGLIO.agda</td>
<td>Datatype <em>[</em>,<em>]</em> ⧵ <em>[</em>,<em>]</em></td>
</tr>
<tr>
<td>Value and heap</td>
<td>Store.agda</td>
<td>Datatype Location, Value, Cell, and Store Location, Value, Cell, and Store</td>
</tr>
<tr>
<td>Machine configuration</td>
<td>Interp.agda</td>
<td>Datatype Conf and Result</td>
</tr>
<tr>
<td>Interpreter (figure 8, 3, 4, and 5)</td>
<td>Interp.agda</td>
<td>Function lookup</td>
</tr>
<tr>
<td>Well-typed environment</td>
<td>WellTypedness.agda</td>
<td>Datatype <em>[</em>,<em>]</em> ⧵ <em>[</em>,<em>]</em></td>
</tr>
<tr>
<td>Well-typed heap (store)</td>
<td>WellTypedness.agda</td>
<td>Datatype <em>[</em>,<em>]</em> ⧵ <em>[</em>,<em>]</em></td>
</tr>
<tr>
<td>Well-typed result</td>
<td>WellTypedness.agda</td>
<td>Datatype <em>[</em>,<em>]</em> ⧵ <em>[</em>,<em>]</em></td>
</tr>
<tr>
<td>Variable lookup lemma</td>
<td>WellTypedness.agda</td>
<td>Function lookup-safe and lookup-safe-corollary</td>
</tr>
<tr>
<td>Heap lookup lemma</td>
<td>WellTypedness.agda</td>
<td>Function lookup-safe and lookup-safe-corollary</td>
</tr>
<tr>
<td>castT’ is well-typed (lemma)</td>
<td>WellTypedness.agda</td>
<td>Function castT’</td>
</tr>
<tr>
<td>Heap update lemma</td>
<td>WellTypedness.agda</td>
<td>Function ext-update-pres−‘v</td>
</tr>
<tr>
<td>Heap new lemma</td>
<td>WellTypedness.agda</td>
<td>Function ext-new-pres−‘v</td>
</tr>
<tr>
<td>V is type safe (prop 1)</td>
<td>InterpSafe.agda</td>
<td>Function V-safe</td>
</tr>
<tr>
<td>Type safe (theorem 2)</td>
<td>TypeSafety.agda</td>
<td>Function type-safety</td>
</tr>
<tr>
<td>Examples in listing 3</td>
<td>Example.agda</td>
<td>Module FunExample</td>
</tr>
<tr>
<td>Examples in listing 4</td>
<td>Example.agda</td>
<td>Module LabExample</td>
</tr>
<tr>
<td>Examples in listing 5</td>
<td>Example.agda</td>
<td>Module RefExample</td>
</tr>
</tbody>
</table>