LJGS: Gradual Security Types for Object-Oriented Languages

Luminous Fennell[1] and Peter Thiemann[2]

1 **University of Freiburg**
Georeges-Köhler Allee 79, 79110 Freiburg, Germany
`fennell@informatik.uni-freiburg.de`

2 **University of Freiburg**
Georeges-Köhler Allee 79, 79110 Freiburg, Germany
`thiemann@informatik.uni-freiburg.de`

Abstract

LJGS is a lightweight Java core calculus with a gradual security type system. The calculus guarantees secure information flow for sequential, class-based, typed object-oriented programming with mutable objects and virtual method calls. An LJGS program is composed of fragments that are checked either statically or dynamically. Statically checked fragments adhere to a security type system so that they incur no run-time penalty whereas dynamically checked fragments rely on run-time security labels. The programmer marks the boundaries between static and dynamic checking with casts so that it is always clear whether a program fragment requires run-time checks. LJGS requires security annotations on fields and methods. A field annotation either specifies a fixed static security level or it prescribes dynamic checking. A method annotation specifies a constrained polymorphic security signature. The types of local variables in method bodies are analyzed flow-sensitively and require no annotation. The dynamic checking of fields relies on a static points-to analysis to approximate implicit flows. We prove type soundness and non-interference for LJGS.

1998 ACM Subject Classification D4.6. Security and Protection

Keywords and phrases gradual typing, security typing, Java, hybrid information flow control

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.9

Supplementary Material ECOOP Artifact Evaluation approved artifact available at
`http://dx.doi.org/10.4230/DARTS.2.1.4`

1 Introduction

Information-flow control (IFC) is a cornerstone of language-based security. A typical IFC policy rules out the flow of information from classified sources to public sinks. The technical property aimed for is noninterference: changing a classified source does not influence the public sinks. Noninterference comes in different flavors depending on the observational capabilities of an attacker (e.g., termination sensitive or not, batch or interactive).

There are also different kinds of IFC. A static system performs a static analysis, for instance using a security type system, and guarantees noninterference for analyzed programs. A dynamic system attaches run-time security labels to values, propagates them along with the values, and checks them at appropriate points during program execution. Hybrid systems employ additional static analysis to improve precision in the detection of implicit flows [23].

The choice between a static and a dynamic system is a difficult trade-off in the development of secure software. For new software components that are designed with security in mind and for components that have high reliability requirements a static analysis is a good fit: It does not impose any run-time overhead and provides hard static guarantees whereas a dynamic system needs to manage security labels at run-time and often aborts a running program when a security problem is detected.

For the integration of legacy components as well as for prototyping, static IFC can impede programming productivity significantly: Simple analyses yield many false positives and thus may reject many programs that are in fact secure. Precise analyses often require extensive manual annotation of the source code. They rely on complex abstract domains that may be computationally expensive or difficult to understand by non-experts. In contrast, dynamic approaches rely on an instrumented run-time system. Thus, after specifying a security policy, a programmer can debug policy breaches by testing and inspecting concrete program runs.

Contributions

To make the benefits of both approaches available to mainstream object-oriented programming, we propose a type-based amalgamation of static and dynamic IFC for Java. Inspired by work on gradual typing [26, 14], our system enables programmers to rely on the guarantees and efficiency of a lightweight security type system and to back off to dynamic checking when necessary. Gradual typing enables the composition of programs from typed and untyped fragments using suitable type casts at the interfaces. It guarantees full compliance with the type system in the typed fragments, whereas untyped fragments may raise run-time type errors. *Gradual security typing* transposes this approach to an information flow setting.

Lightweight Java with Gradual Security (LJGS) is an object-oriented core calculus that implements our ideas. LJGS is a subset of Java without threads, exceptions, or reflection. It is inspired by Lightweight Java (LJ) [29]. Our specification of LJGS is type-based: it takes the form of a type system and it assumes that the underlying program is well-typed. Thus, LJGS is independent of specific Java typing features like generics. The LJGS types in this paper amount to type annotations in an actual Java program.

Following Denning [10], LJGS specifies permissible information flows using a lattice $(Sec, \sqsubseteq)$ of *security levels*. For each field of a class, the programmer specifies a fixed security level *or* marks the field as dynamically checked. For each method, the programmer supplies a constrained polymorphic security type signature (see examples in Section 2). The signature relates the security types of the arguments, the result, and the effect on global variables with each other and with fixed levels that arise from field accesses. The security types of local variables need *not* be declared. Moreover, the types of local variables are *flow sensitive*, that is they can change during the method; object field types are *flow insensitive* to keep static checking light-weight. In contrast to other work on security type signatures for Java [30] our type checking algorithm works directly on the constraints.

The LJGS type system is designed so that the security level of a value with a static security type does not have to be tracked at run-time. Each value with a dynamic security type, in a dynamic field or in a temporarily dynamic local variable, carries a run-time security label that overapproximates its true security level. Implicit flows are detected with an approach inspired by *hybrid monitors* [23]: at the join points in the control-flow graph, security labels of variables and fields are upgraded preemptively if they could be updated in the untaken branches. A points-to analysis assists in determining the heap references to fields that require upgrading. Purely dynamic approaches like the no-sensitive-upgrade (NSU) policy [3, 36] may be used as well, but with more conservative results.[1]

[1] We used NSU in a previous version of this paper: `http://proglang.informatik.uni-freiburg.de/projects/gradual/ljgs/fcs2015.pdf`

No existing system provides a similarly powerful combination of static and dynamic IFC for a Java-like language. In particular, prior work [12, 13] illustrates the principles of gradual security typing with toy languages. While LJGS is the result of extrapolating similar principles to an object-oriented setting, extensions required in practice, like polymorphic signatures and avoiding run-time checks for static code, yield a type- and run-time system that differs significantly from prior designs. The differences are further elaborated in Section 9.

Technical results: Besides proving type preservation and progress (up to breaches of the security policy in dynamic updates), our key result is *batch termination insensitive non-interference* (BTINI), as defined by Askarov et al [2]. We assume that an attacker can only access the public part of the heap, may construct the public arguments of any LJGS method, run it, and inspect only the public part of the result and the heap afterwards. If the method diverges or aborts, the attacker receives no information. There is no way to obtain intermediate results.

Scope of the LJGS calculus

LJGS supports the features of class-based, typed object-oriented languages that pose fundamental challenges for information flow control: mutually recursive methods and classes, mutable local variables, mutable objects, and virtual method calls. In the following, we summarize these challenges and then discuss features omitted from LJGS, namely exceptions, and Java-style downcasts and type-tests. Furthermore, Section 7 discusses how a practical language based on LJGS could deal with reflective and otherwise hard-to-analyze code. **Mutually recursive classes and methods** complicate global, inter-procedural inference of types and information flow. LJGS supports local type inference, only. It avoids the need for global inference with annotations on methods and classes. The issue of **mutable local variables** arises already in simple imperative languages [24], but it is not addressed by previous work on gradual security [12, 13]. For LJGS, we adapt best practice in maintaining a *program counter type* to track implicit flows and in handling variables flow-sensitively [17, 23] to improve precision.

Mutable objects may lead to *global* implicit flows, as the following example shows:

```
class C { int F; void setF(){ this.F = 1; };}
C c = new C(); C d = c;
if (secret == 42) {d.setF();}
printPublic(c.F); // <- information leak
```

Here, `secret` has a high-security type and `printPublic` is a low-security sink. The field `F` of the object `c` is updated under a high-security program counter by calling method `setF()` on `d` which is an alias of `c`. Afterwards, `c.F` is publicly exposed by `printPublic()`. However, there is no local indication for the leak: the program does not directly change `c.F`. The reason for the leak is that `c`, `d`, and **this** in the method call to `setF` are aliases for the same object during the execution of the program. Sound information flow control has to prevent low-security access to the field `c.F` after the update, even if the update is performed on a different alias or in a different method.

In LJGS and other security-typed languages [21, 22] this requirement is enforced by fixed, flow-insensitive security types for fields and by write-effect annotations on methods that indicate the types of fields that a method updates. For static, high-security program counters, low-security or dynamic effects are forbidden. Dynamic program counters admit only dynamic effects and dynamic fields updates. LJGS treats mutable objects differently than previous work[13], where run-time labels are required even for statically typed object references.

Virtual method calls cause conditional control flow and thus may create implicit information flows: the run-time type of the receiver object selects the implementation of a method to execute. While dynamic enforcement of information flow copes naturally with dynamic dispatch, statically checking a virtual method requires a sound, static approximation of the security properties of all possible implementations that might be executed. LJGS supports dynamic dispatch for Java-like class derivation and method overriding. To soundly type-check method calls, the type system imposes a restriction on signatures of overriding methods (cf. Definition 5, Section 4.4).

Among the features that LJGS omits, support for (catchable) **exceptions** is the most challenging for information flow control:[2] throwing an exception introduces non-local implicit information flows from the program counter of the throw-site to any exception handler in the call stack. Practical languages with type-based support for information flow control like JIF [21] and FlowCaML [22] solve this problem with additional effect annotations on methods that track the program counter type at the throw-site. This exception effect is an appropriate lower bound for the program counter type of exception handlers. To simplify the presentation in this paper, we do not consider catchable exceptions, but we expect the extension of LJGS's type system with exception effects to be unsurprising, as the typing constraints of LJGS are similar to FlowCaML's constrained type schemes.

Banerjee and Naumann identify (down-)**casting of classes and type test** as potential sources of information flow [6]. A conditional based on a type test creates an information flow from the tested object to the program counter of the branches. For example, the body of the condition in line 5 executes with a high-security program counter, as the run-time class of `c` depends on the value of the high-security variable `secret` (line 4).

```
1  class C { }
2  class D extends C { }
3  C c;
4  if (secret == 42) { c = new D(); } else { c = new C(); }
5  if (c instanceof D) {
6   ... // <-- high-security program counter
7  }
```

LJGS does not support type tests directly, but as the objects involved in branch conditions are already tracked, the feature would be straightforward to add as a primitive operation. Downcasts may result in additional exceptional flows when a cast error is recoverable. As we currently do not consider exceptions for LJGS, we also omit downcasts for simplicity.

2 A Taste of LJGS

This section demonstrates the salient features of LJGS. The security lattice used in the examples has at least two points, `LOW` and `HIGH`, where `LOW` $\sqsubseteq$ `HIGH` and `HIGH` $\not\sqsubseteq$ `LOW`.

Figure 1 illustrates how to specify information flow policies via security signatures. The method `max` calculates the maximum of its parameters `x` and `y`. Clearly, the method's result depends on both parameters. To express this dependency, method `max` carries polymorphic constraints in the **where** clause of its signature. It indicates that information may flow from `x` and `y` to the result by asserting that the parameters' security types are lower bounds for the type of the return value. The variables `x`, `y`, and `ret` stand for the security type of

[2] It is unproblematic to extend LJGS with *uncatchable* exceptions, as abrupt program termination does not violate BTINI.

```
1   int max(int x, int y) where { x ≤ ret, y ≤ ret } {
2     if (x ≤ y) {x = y;} return x;
3   }
4
5   class Logger { String[LOW] buf; String[HIGH] hbuf; String[⋆] dbuf; }
6
7   int maxMsg(Logger log, int x, int y)
8     where { x ≤ ret, y ≤ ret , log ≤ LOW} and { LOW } {
9     if (x ≤ y) { x = y; }
10    log.buf = "max was called";
11    return x;
12  }
```

Figure 1 Examples of method definitions and polymorphic signatures.

```
1   Logger log = new Logger(); int x = (⋆ ⇐ HIGH) secret;  int y = (⋆ ⇐ LOW) 42;  int r;
2
3   r = this.maxMsg(log, secret, 42);
4   r = this.maxMsg(log, x, y);
5
6   // type error: uncontrolled mix of dynamic and static arguments
7   // r = this.maxMsg(log, secret, y);
8
9   if (x == 42) {
10    // type error: program counter is statically unknown
11    // r = this.maxMsg(log, x, y);
12  }
```

Figure 2 Calling polymorphic methods.

the parameters `x`, `y`, and the return value. As the variables are not bounded by concrete security levels, the signature is polymorphic and `max` may be called under any program counter type with arguments that satisfy the constraints. It may also be called with two dynamic arguments.

Line 5 of Figure 1 defines a class `Logger` with three fields: `buf` has with static security type `LOW`, `hbuf` with static security type `HIGH`, and `dbuf` with dynamic type ⋆. The method `maxMsg` is a version of `max` with global side effects; it writes to the low-security field of a `Logger` object. The signature of the method indicates this global effect to a `LOW` field in the **and** part of the **where** clause. LJGS requires that the *program counter type* at all call sites is smaller than the declared effect `{LOW}`, which guarantees that `maxMsg` is only called securely in contexts where the program counter security level is `LOW`. The signature of `maxMsg` additionally enforces that the `log` parameter is a low-security reference by upper-bounding it with `LOW`. The parameters `x` and `y` remain polymorphic, as they do not influence the global side effect. Typically, parameters have concrete security types as upper bound, like `log` in method `maxMsg`, if either the method requires *read or write access* to a field of that type or if the parameter flows into a program counter for a field update.

Figure 2 illustrates the use of polymorphic methods in dynamic and static code. Dynamically checked values are created with a *value cast* like `(⋆ ⇐ HIGH)secret`. The two casts in line 1 create a dynamic version of the high-security value in variable `secret` and a low-security constant `42`, storing them in `x` and `y`, respectively. The *source type* of a cast, here `HIGH` and `LOW`, respectively, indicates what run-time label to attach to the cast value. The *destination type* is the dynamic type, ⋆. The polymorphic method `maxMsg` may be called with arguments that are either all static (line 3) or all dynamic (4). The recipient `r` is typed flow-sensitively and has type `HIGH` in line 3 and type ⋆ in line 4.

Calling `maxMsg` with mixed static and dynamic arguments, as shown in line 7, is not allowed in this example: The dynamics of LJGS (cf. Sections 5 and 6) do not represent static

```
 1 int maxMsgDyn(Logger log, int x, int y)
 2     where { x ≤ ret, y ≤ ret, log ≤ ⋆} and { ⋆ } {
 3   if (x ≤ y) { x = y; }
 4   log.dbuf = "max was called"; return x; }
 5
 6 void logResults(Logger log, boolean privMode, int h, int l)
 7   where { HIGH ≤ h , l ≤ LOW,
 8          , privMode ≤ ⋆, privMode ≤ LOW , log ≤ ⋆, log ≤ LOW}
 9    and { ⋆, LOW } {
10  log.dbuf = "public result:" + (⋆ ⇚ LOW) l;
11  if (privMode) {log.dbuf += "secret result:" + (⋆ ⇚ HIGH) h;  }
12  /* ... */
13  if (!privMode) {log.buf = (LOW ⇚ ⋆) log.dbuf;  }
14 }
15
16 boolean high = ...; // a high-sec. value
17 maxMsgDyn(log, x ,y);
18 if (high) {(HIGH ⇛ ⋆) {  maxMsgDyn(log, x ,y); }}
```

Figure 3 Examples of methods using dynamic IFC.

security information at run-time. Thus, information flow between static and dynamic types cannot be controlled only by inspecting run-time security labels and the type system forbids flows from static to dynamic entities except when they are explicitly controlled by casts. Uncontrolled flows from dynamic to static entities are also forbidden, as dynamic labels are not available at type-checking time. To enforce both restrictions, constraints of the form $x \leq y$ are not satisfiable if x is static and y is dynamic, or vice-versa. Thus the call in Line 7 violates the signature of `maxMsg` which relates both arguments with **ret**.

LJGS imposes similar restrictions for implicit flows. For example, the call to `maxMsg` in line 11 would be rejected. Testing the dynamic variable creates a dynamic program counter that cannot be checked against the static effect of `maxMsg`'s signature.

Figure 3 illustrates the distinguishing feature of LJGS: the integration of dynamic and static enforcement of security policies.

The method `maxMsgDyn` is a less restrictive version of `maxMsg` that relies on dynamic IFC. As before, its signature states that `x` and `y` flow into the return value. But instead of a `LOW` effect, it asserts a dynamic effect and requires that the `log` object admits dynamic access (i.e., is smaller that $\star$). While `maxMsg` writes to a low-security field and is thus incompatible with high-security program counters, `maxMsgDyn` uses the dynamic field `dbuf`. Dynamic fields are checked flow-sensitively at run-time, so that LJGS permits calls to `maxMsgDyn` in low- and high-security program counters (lines 16 to 18 in Figure 3). After line 17, `log.dbuf` contains a dynamic low-security string. The statement starting with "`(HIGH` $\Rrightarrow\star$`)`" in line 18 is a *context cast* which instructs the run-time system to propagate a high-security program counter label for the statements contained in the cast. As a result, the dynamic label of `log.dbuf` can be updated to a high-security label after the call in line 18 to reflect the implicit flow from the program counter into `log.dbuf`.

The method `logResults` (also in Figure 3) shows how to adapt information flow behavior dynamically using a boolean flag. It takes as arguments a `log` object, a boolean flag `privMode` and two integers `h` and `l`. The method's purpose is to write the values of `l` and `h` to various buffers in the `log` object, depending on the flag `privMode` ("*private mode*"). The signature states that `h` is `HIGH`[3] and `l` is `LOW`. It also states that the method has both, an effect of type dynamic ($\star$) *and* static-low (`LOW`). Similarly it requires `log` to admit accesses to fields of type

[3] A concrete security type as lower bound to a parameter is never required in LJGS. Here it is specified to make `h` a high-security value for the sake of the example.

```
interface Modifier { int modify(int x) where { x ≤ ret }; }

int maxMod(int x , int y, Modifier ymod) where { x ≤ ret, y ≤ ret, ymod ≤ ret }{
   z = ymod.modify(y);
   if (x ≤ z) { return z } else { return x }
}
```

Figure 4 Example of object-oriented code.

$\star$ and `LOW`, and that `privMode` can flow into fields of type $\star$ and `LOW`. The reason for these constraints become clear when inspecting the method's body: in line 11 there is a dynamic write of a dynamic high-security value to `dbuf` in the context of `privMode` and `log` and in line 13 there is a static write to `buf` using the value of `dbuf` cast to `LOW` in the context of `privMode` and `log`. Despite the copy of the potentially high-security content of `dbuf` to `buf` in line 13, the method is actually secure: the flag `privMode` guards the field updates such that a high-security content of `dbuf` is never written to `buf`. If the author of `logResults` would have failed to guard the field updates correctly, for example by accidentally omitting the negation of `privMode` in line 13, the value cast from $\star$ to `LOW` would have aborted the program by signalling a dynamic security violation. Using boolean flags in such a way is only possible in the dynamic fragment of LJGS; if `log.dbuf` were static, the type system would require it to have a high-security type and unconditionally forbid the copy to `log.buf`.

It remains to explain how the constraints for `privMode` and `log` can be possibly satisfied, given the premise that a value cannot be implicitly converted between static and dynamic types. LJGS includes a special bottom type, called the *public type* ($\bullet$), for this purpose. The type systems ensures that such a public value carries no confidential information so that it can be used in dynamic code without carrying a run-time label.

Figure 4 shows a final example that illustrates the interaction of LJGS' gradual security typing with object-oriented code. The interface `Modifier` specifies a method `modify` with a signature that asserts no global side effects and where the input flows into the output. The method `maxMod` calculates the maximum of its arguments `x` and `y` after modifying `y` using the `Modifier` object `ymod`. Its signature contains the same constraints as `max` in Figure 1 and adds the additional requirement that `ymod` is less confidential than the result.

The following code snippet implements the polymorphic `max` method with `maxMod`:

```
class Id extends Modifier {int modify(int x) where { x ≤ ret } { return x; }}
// re-implementation of max
int max2(int x, int y) where {x ≤ ret, y ≤ ret} { return maxMod(x, y, new Id()); }
```

Class `Id` implements an identity modification. Its `modify` method inherits the constraints of the `Modifier` interface. In LJGS, freshly create objects are always public. By passing a new `Id` instance to `maxMod`, `max2`'s signature is as flexible as that of `max`: the constraint `ymod` $\leq$`ret` of `maxMod` is trivially satisfied and can be omitted from the signature of `max2`.

Semantically, the result of `max2`, or of `max` for that matter, always has a security level that is the least upper bound of `x` and `y`. Thus, calling `max2` with statically or dynamically typed arguments makes little difference. In the following call to `maxMod`, however, a static call is more conservative than a dynamic one.

```
class Erase extends Modifier {int[•] def; int modify(int y) { return def; }}
/* ... */ z = maxMod(42, y, new Erase(0)); /* ... */
```

Here we use an `Erase` modifier that replaces the value of `y` value with a public constant. The signatures of `maxMod` and `Erase.modify` still assert a dependency on the argument `y` but dynamically the connection is cut by the particular implementation of `Erase.modify`. Thus,

$$
\begin{array}{lll}
prog & ::= cld_1 \mathinner{..} cld_n\, s & \\
cld & ::= \textbf{class}\, C\, \textbf{extends}\, cl\{F_1[a_1] \mathinner{..} F_n[a_n]\; md_1 \mathinner{..} md_m\} & cl ::= C \mid \textbf{Object} \\
a & ::= \bullet \mid A \mid \star \quad md ::= M(var_1, \mathinner{..}, var_n)\, \textbf{where}\, \mathcal{S}\, \textbf{and}\, \mathcal{G}\{s\, \textbf{return}\, y\} & \\
x, y & ::= var \mid \textbf{this} & \\
e & ::= x \mid x.F \mid N \mid x + y \mid \textbf{new}\, C(x_1, \mathinner{..}, x_n) \mid (a \Lleftarrow a')x & \\
s & ::= var = e \mid x.F = y \mid var = x.M(y_1, \mathinner{..}, y_n) \mid s;\ s & \\
 & \mid\ \textbf{if}\,[sl, \mathcal{L}, \mathcal{G}](x == y)\{s\}\{s\} \mid \textbf{while}\,[sl, \mathcal{L}, \mathcal{G}](x == y)\{s\} \mid (a \Rrightarrow a')\{s\} & \\
sl & ::= \text{unique identifiers for branching statements} &
\end{array}
$$

Figure 5 Syntax of LJGS.

the following call succeeds without security problems when `printPublic` is a low-security sink and `secret` a static high-security variable.

```
y = (⋆ ⇚ HIGH) secret; z = maxMod(42, y, new Erase(0)); printPublic((LOW ⇚ ⋆)z);
```

A corresponding static call without the typecasts would be rejected by the type checker: `maxMod`'s signature would type `z` as `HIGH`, even though no information flows from `y` to `z`.

3 Syntax of LJGS Programs

LJGS extends Lightweight Java with annotations for security types and casts.

To better focus on the security aspects and to avoid notational clutter, we omit the Java types and signatures for fields, local variables, and methods; we only write the annotations relevant for security typing.[4] Nevertheless, we assume that all programs are well-typed Java programs where types, signatures, and declarations of local variables are erased.

An LJGS program (see Figure 5) consists of a set of class definitions, *cld*, followed by a statement s, the entry point to the program. Class definitions consist of field declarations, $F_i[a_i]$, and method declarations, md_i. Classes form a hierarchy with **Object** at its root. For simplicity, all method and field names are unique and we assume that the relationship of method overrides is externally defined.

A field declaration relates a field name with a security type a. A security type is either the *public type* $\bullet$, a static security level A, or the *dynamic type* $\star$.

A method definition declares the method name, M, a list of parameters, a set of *method constraints* $\mathcal{S}$, the global effect $\mathcal{G}$, a statement s that serves as method body, and a single local variable y for the return value. We write $\text{params}\,(M)$, $\text{constraints}\,(M)$, and $\text{effects}\,(M)$ to refer to the parameters, method constraints and effects, respectively. Method constraints and effects are discussed in detail in Sections 4.1 and 4.3. Further local variables are implicitly defined by their first assignment.

A variable x is either user-defined, var, or references the receiver of the method call, **this**. An expression e can be a variable access, x, field access $x.F$, an integer constant, N, integer addition, $x + y$, object instantiation, $\textbf{new}\, C(x_1, \mathinner{..}, x_n)$, or a value cast. A cast expression $(a \Lleftarrow a')x$ converts the value of x from *source type* a' to *destination type* a. We omit a **null** value for simplicity. It would be straightforward to add, however, as **null** behaves like an integer constant with respect to information flow.

[4] In an implementation, we have to write Java types and signatures as well as security annotations.

$$\begin{array}{lcl} \mathcal{S} & ::= & \{sc_1, .., sc_n\} \\ sc & ::= & st \leq st \mid st \sim st \\ st & ::= & a \mid x \mid \mathbf{ret} \end{array} \qquad \begin{array}{lcl} \mathcal{C} & ::= & \{c_1, .., c_n\} \\ c & ::= & sec \leq sec \mid sec \sim sec \\ sec & ::= & st \mid \alpha \\ tvar & ::= & x \mid \mathbf{ret} \mid \alpha \end{array}$$

Figure 6 Components of method constraints and typing constraints.

A *local update*, $var = e$ assigns the value of an expression e to a variable. A *field update* $x.F = y$ writes the value of y to field F of the object referenced by x. A *method call* $var = x.M(y_1, .., y_n)$ stores the result of calling method M with arguments $y_1, \ldots, y_n$ in variable var. The **if** and **while** statements are standard except for three annotations, a *source location sl* and *effects* $\mathcal{L}$ and $\mathcal{G}$. The role of the effects is further explained in Sections 4.3 and 5.4. The external write effect analysis that backs the dynamic IFC uses source locations to identify branching statements. The role of the effect analysis is further explained in Section 5.3. The examples of Section 2 do not show locations or effects on **if** statements as they can be inferred. Finally, the *context cast* statement $(a \Rrightarrow a')\{s\}$ embeds a computation with *inner* program counter type a' into a context with *outer* program counter type a.

4 Security Constraints and Typing Rules

In LJGS, a programmer specifies information-flow properties by providing suitable method signatures. As illustrated in Section 2, a signature relates parameters, return values, and side effects of methods with security types. The type system ensures that the signature specifications are adequate and that operations depending on statically typed parameters and fields have no security leaks. Its design also ensures that the security levels of statically typed values need not be tracked at run-time.

Security types form a lower semi-lattice induced by the following ordering.

▶ **Definition 1** (Partial order on security types).

$$\bullet \leq a \qquad\qquad A \leq A' \text{ if } A \sqsubseteq A' \qquad\qquad \star \leq \star$$

By embedding the lattice of security levels into security types, LJGS supports the usual notion of security subtyping: values with a low security level are implicitly promoted to a higher one and contexts with a low-security program counter type admit computations that perform side effects on a higher security level. To ensure a clean boundary between static and dynamic code, implicit conversion between static types and the dynamic type is not allowed; that is, the supremum of A and $\star$ is not defined. However, the public security type $\bullet$ may act polymorphically as the dynamic type and the static type at the bottom of the security lattice ($\perp$).

4.1 Method Constraints

The constraints in a method signature represent the information flow dependencies between parameters and return values: flows into the return value are represented by lower bounds on a return symbol, whereas flow restrictions on the arguments are represented by upper bounds on parameter symbols. Figure 6 defines the syntax for method constraints $\mathcal{S}$. A constraint relates its *method constraint types*, st_1 and st_2, either by *subsumption*, $st_1 \leq st_2$, or by *compatibility*, $st_1 \sim st_2$. Method constraint types st are literal security types a or symbols for a method's formal parameters x (including **this**) and return value, **ret**. For simplicity, we require that all method constraints mention exactly the parameters of their respective methods, as well as **ret** and **this**.

The constraint $st_1 \leq st_2$ ("st_2 subsumes st_1") specifies that st_1 and st_2 should be ordered according to Definition 1. For example, the constraint `HIGH` $\leq$ **ret** requires the return value of a method to be a statically known security level that is at least `HIGH`. The constraint $st_1 \sim st_2$ ("st_1 is compatible with st_2") specifies that if st_1 is dynamic then st_2 should be dynamic or public and vice versa. If both components are static, then they are compatible.

Compatibility constraints appear rarely in signatures, but here is an example. Consider a method, `const42`, that contains the same statements as `max`, but subsequently cuts the flow from parameters to the result by overwriting the `x` with a constant.

```
int const42(int x, int y) where { x ∼ y } {
  if (x ≤ y) {x = y;}
  x = 42; return x;
}
```

Although `const42` ignores the parameters and does not constrain `ret`, the parameters `x` and `y` cannot have arbitrary types: If `const42` is called with a dynamic first argument, the second argument needs to be dynamic or public to enable the run-time system to track implicit flows arising from the conditional. The compatibility constraint $x \sim y$ expresses this requirement.

4.2 Typing Constraints and Constraint Interpretation

To type-check a method's body against its signature, LJGS' typing rules (cf. Section 4.4) generate sets of *typing constraints*, $\mathcal{C}$. Typing constraints c (see Figure 6) are like method constraints but relate *statement constraint types*, *sec*, which extend method constraint types with *local type variables* α. Local type variables represent the flow-sensitive security types for the local variables before and after the statements of a method body. A method body complies to a signature if the generated typing constraints refine the information flow between parameters, fields and return value that the signature represents.

We now formalize the intuitive notion of constraints that we used in the examples in Section 2 and Section 4.1. In the following, we let $\mathcal{C}$ range over sets of typing constraints, α, β, γ over local type variables, and we write $\alpha^\dagger$ for a fresh local type variable. We also let *tvar* range over statement constraint types that are not literal security types (cf Figure 6).

▶ **Definition 2** (Assignment, solution, solvability)**.** An assignment is a total function from type variables to security levels. An assignment θ is extended to a total function from components to security levels, θ^*, by mapping security levels to themselves:

$$\theta^*(sec) = a \quad \text{if } sec = a \qquad\qquad \theta(tvar) \quad \text{if } sec = tvar$$

An assignment θ is a *solution* of a constraint set $\mathcal{C}$, written $\theta \models \mathcal{C}$, iff (i) for all constraints $sec_1 \leq sec_2 \in \mathcal{C}$, it holds that $\theta^*(sec_1) \leq \theta^*(sec_2)$, and (ii) for all constraints $sec_1 \sim sec_2$, it holds that there exists a type a such that $\theta^*(sec_1) \leq a$ and $\theta^*(sec_2) \leq a$.

A constraint of the form $\bullet \leq sec$ imposes no restriction on solutions. A compatibility constraint that contains a static and a dynamic component, like $\star \sim$ `LOW`, is never solvable.

Type checking of LJGS requires that the typing constraints generated for method bodies refine the constraints of the respective signatures. For method- and typing constraints, refinement amounts to checking entailment of constraints modulo local type variables.

▶ **Definition 3** (Refinement of typing constraints)**.** Let $\mathcal{C}_1$, $\mathcal{C}_2$ be two constraint sets. $\mathcal{C}_2$ *refines* $\mathcal{C}_1$, written $\mathcal{C}_2 <: \mathcal{C}_1$, if for each solution $\theta \models \mathcal{C}_1$ there exists an assignment θ' such that (i) $\theta' \models \mathcal{C}_2$, (ii) $\theta'(x) = \theta(x)$ for all parameters x, and (iii) $\theta'(\mathbf{ret}) = \theta(\mathbf{ret})$.

Example: the constraints $\{\texttt{LOW} \leq \mathbf{ret}\}$ and $\{\texttt{LOW} \leq \alpha, \alpha \leq \mathbf{ret}, \texttt{HIGH} \leq \beta\}$ both refine the (more restrictive) constraint $\{\texttt{HIGH} \leq \mathbf{ret}\}$.

4.3 Effects

Typing of LJGS method bodies includes the generation of two kinds of effects, $\mathcal{L}$ and $\mathcal{G}$. The *local effect* $\mathcal{L}$ is a set of variables. It appears as annotation on branching statements. If a local variable is listed in $\mathcal{L}$ then the branching statement may modify that variable and thus potentially taint it with information from the branch condition. The run-time system uses $\mathcal{L}$ to update local variables in untaken branches.

The *global effect* $\mathcal{G}$ is a set of security types. It appears on method definitions and branching statements. If a security type is listed in $\mathcal{G}$, then the method or branching statement may perform a side effect that leaks information into a (globally) accessible field of that type. The type system uses $\mathcal{G}$ to track implicit flows to the heap.

There is a refinement relation for global effects that amounts to checking subsumption contravariantly on the contained security types.

▶ **Definition 4** (Subsumption of global effects)**.** $\mathcal{G}_2$ *refines* $\mathcal{G}_1$, written $\mathcal{G}_2 <: \mathcal{G}_1$, if for all $a \in \mathcal{G}_2$ there exists $a' \in \mathcal{G}_1$ such that $a' \leq a$.

For example, the global effects $\{\mathit{high}, \star\}$ and $\{\mathit{low}, \mathit{high}, \star\}$ both refine $\{\mathit{low}, \star\}$. In contrast $\{\mathit{high}, \star\}$ does note refine $\mathcal{G} = \{\mathit{low}\}$, as there is no type in $\mathcal{G}$ that is smaller or equal to $\star$.

4.4 Typing Rules

We are now in a position to define the rules for LJGS. In addition to the typing requirements for Java-like languages, well typed LJGS programs satisfy three conditions: all method constraints are satisfiable, the signatures of overriding methods of every subclass refine the corresponding signatures of its superclass, and, for every method, LJGS' statement typing rules generate constraints and effects that refine the method's signature.

The following definition captures the refinement requirement for subclassing:

▶ **Definition 5** (Well-typed class hierarchy)**.** Let constraints (M) and effects (M) be the constraints and the effects that are given by the signature of method M. Method M_1 *refines* method M_2 if constraints $(M_1) <:$ constraints (M_2) and effects $(M_1) <:$ effects (M_2).

The class hierarchy of an LJGS program is well-typed if for every method M_1 that overrides a method M_2 method M_1 also refines method M_2.

The following judgment summarizes the requirements for method bodies:

$$\frac{\begin{array}{c}\Gamma_1 = [\mathit{var}_1 \mapsto \mathit{var}_1, .., \mathit{var}_n \mapsto \mathit{var}_n, \mathbf{this} \mapsto \mathbf{this}] \\ \gamma \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, (\mathcal{L}, \mathcal{G}') \qquad \alpha = \Gamma_2(y) \qquad \mathcal{C} \cup \{\alpha \leq \mathbf{ret}\} <: \mathcal{S} \qquad \mathcal{G}' <: \mathcal{G}\end{array}}{\vdash M(\mathit{var}_1, .., \mathit{var}_n)\,\mathbf{where}\,\mathcal{S}\,\mathbf{and}\,\mathcal{G}\{s\,\mathbf{return}\,y\}}$$

Its central requirement is the judgment $\gamma \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}$ which generates constraints $\mathcal{C}$ and effects $\mathcal{E}$ for well-typed statements. Effects $\mathcal{E}$ have the form $(\mathcal{L}, \mathcal{G})$. Γ_1 and Γ_2 are typing environments that map local variables to the type variables. The environments connect a local variable with the variable that represents its type before and after a particular statement. Together with $\mathcal{C}$, the *initial environment* Γ_1 describes the typing constraints for local variables before s is executed, whereas the *final environment* Γ_2 describes local variable types after executing s. The type variable γ represents the type of the program counter. The method typing judgment pre-initializes the environment for the parameters and checks if the generated constraints and effects refine the constraints and effects of the signature.

4.4.1 Statement Typing: Overview

Section 4.4.2 explains some of the typing rules in detail. A complete listing of rules can be found in a technical report[5]. Although the details of the typing rules may seem complex the basic principles behind the constraint generation are rather simple: To track explicit flows in assignment statements, the rules generate constraints where each read variable or field is a lower bound to the variable written. For example, the statement `x = z.F; x = x + y` yields constraints that include $\mathcal{C} = \{z \leq \beta_1, a \leq \beta_1, \beta_1 \leq \beta_2, y \leq \beta_2\}$. Here, a is the type of the field F, z and y are parameters and β_1, β_2 are the type variables that the environment maps to x after the first and second assignment, respectively. The typing rules consider implicit flows by checking with the program counter variable γ. Each assignment generates an additional constraint where the program counter variable is a lower bound of the updated variable. In the example, $\mathcal{C}$ would be extended with the constraints $\gamma \leq \beta_1, \gamma \leq \beta_2$. If, for example, the result of a branch condition flows into the branches of an **if** statement, a new program counter variable is introduced. Program counter variables establish the implicit flows from local variables and parameters to results and field writes. For example, the statement **if** `(x = y) { z.F = 42; }`, where `x, y, z` are parameters and the type of field F is a, and the initial program counter type is γ, generates the constraints $\mathcal{C}_2 = \{\gamma \leq \gamma', x \leq \gamma', y \leq \gamma', \gamma' \leq z, \gamma' \leq a\}$. A method constraint like $\{x \leq z, y \leq z, z \leq a\}$ that subsumes $\mathcal{C}_2$, take the flows through γ' into account without mentioning the program counter variables.

Casts hide information flows from the type system. The constraints generated by assignments with value casts do not connect the type of the right-hand-side variable with that of the result. Instead, they connect the right-hand-side type with the source type of the cast and the cast's destination type with the result. For example, the statement `x = (`$\star \Leftarrow$ `LOW)y` has the constraints $\{\gamma \leq \alpha_1, y \leq \texttt{LOW}, \star \leq \alpha_2\}$ which do not relate the type of `y` with that of `x` (α_1). Similarly to value casts, context casts break up the connection between program counters and insert their respective source and destination types, instead.

4.4.2 Statement Typing: Details

Figure 7 gives the rules for assignments. The rule for local assignment, ST-LOCAL, yields the constraints for a fresh type variable, $\alpha^\dagger$, that represents the type of the updated local variable after the assignment. The final type environment is updated accordingly. An expression typing judgment generates the constraints for explicit flows to $\alpha^\dagger$ while additional constraint $\gamma \leq \alpha^\dagger$ takes care of implicit flows. Writing to variable *var* generates a singleton local effect $\{var\}$. No global effects are generated, as local updates are not visible outside of the method.

Rule ST-PUTFIELD types write operations to a field F. It requires that F's type, indicated by fsec (F), subsumes the type of the source variable, the type of the accessed object reference x, as well as the type of the program counter. The statement's effect is a global effect with F's type. As no local variables are modified, initial and final environments are identical.

[5] `http://proglang.informatik.uni-freiburg.de/projects/gradual/ljgs/ecoop2016-tr.pdf`

$\boxed{\gamma \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}}$

$$\textsc{st-local}\quad \frac{\Gamma_1, \alpha^\dagger \vdash e : \mathcal{C} \qquad \Gamma_2 = \Gamma_1[var \mapsto \alpha^\dagger]}{\gamma \vdash var = e : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C} \cup \{\gamma \leq \alpha^\dagger\}, (\{var\}, \varnothing)}$$

$$\textsc{st-putfield}\quad \frac{\mathcal{C} = \{\Gamma(x) \leq \text{fsec}\,(F), \Gamma(y) \leq \text{fsec}\,(F), \gamma \leq \text{fsec}\,(F)\}}{\gamma \vdash x.F = y : \Gamma \Rightarrow \Gamma, \mathcal{C}, (\varnothing, \{\text{fsec}\,(F)\})}$$

Figure 7 Statement typing: updates.

$$\textsc{rt-new}\quad \frac{\begin{array}{c} a_1 \mathinner{..} a_n = \text{fieldsecs}\,(C) \\ \mathcal{C} = \{\Gamma_1(x_1) \leq a_1, \mathinner{..}, \Gamma_1(x_n) \leq a_n\} \end{array}}{\Gamma, \alpha \vdash \textbf{new}\, C(x_1, \mathinner{..}, x_n) : \mathcal{C}} \qquad \textsc{rt-local}\quad \frac{\mathcal{C} = \{\Gamma(x) \leq \alpha\}}{\Gamma, \alpha \vdash x : \mathcal{C}} \qquad \textsc{rt-cast}\quad \frac{\begin{array}{c} a \lesssim a' \\ \mathcal{C} = \{\Gamma(x) \leq a, a' \leq \alpha\} \end{array}}{\Gamma, \alpha \vdash (a' \Leftarrow a)x : \mathcal{C}}$$

$$\frac{}{\star \lesssim a} \qquad \frac{}{a \lesssim \star} \qquad \frac{}{\bullet \lesssim A} \qquad \frac{}{\bot \lesssim \bullet}$$

Figure 8 Expression typing $\boxed{\Gamma, \alpha \vdash e : \mathcal{C}}$ and castability $\boxed{a \lesssim a'}$.

Figure 8 gives some of the constraint generation rules for expressions. The constraints for object allocation, rule RT-NEW, state that constructor arguments flow into their corresponding field. The meta-function **fieldsecs** yields the security types of a class' fields. The new object reference is considered public on creation. Constants (rule not shown) are also considered public in LJGS and impose no constraints. For variable access, rule RT-LOCAL requires that the result type subsumes the type of the accessed local variable. The rules for addition and field access (not shown) are similar. The rule for casts, RT-CAST, requires that the result type subsumes the destination type of the cast and that the source type subsumes that of the accessed variable. Additionally the source type needs to be *castable* into the destination type, written $a \lesssim a'$. Castability, defined in Figure 8, rules out casts that are either unnecessary or are guaranteed to fail: casts from A to A' trivially succeed when $A \sqsubseteq A'$ and fail when $A \not\sqsubseteq A'$. Casts from A to $\bullet$ also trivially fail when $A \neq \bot$.

Figure 9 shows rules for statements that create new contexts. Rule ST-CALL in Figure 9 covers method calls. It extracts the formal parameters, the method constraints, and the effects of the callee M with the operations params$\,(M)$, constraints$\,(M)$, and effects$\,(M)$. Then, it instantiates the information flows stated in the method constraints for the current context by simultaneous substitution of parameters with arguments and of **ret** with the fresh local type variable $\alpha^\dagger$. Otherwise the rules behave like assignments. The global effect of the method signature is treated analogously as in rule ST-PUTFIELD. The operation $\bigsqcap_\gamma \mathcal{G} := \{\gamma \leq a \mid a \in \mathcal{G}\}$ generates the corresponding constraints. The rule for conditionals, ST-IF, types the branches s_1 and s_2 under a fresh program counter variable $\beta^\dagger$. Types assigned to $\beta^\dagger$ need to subsume the old program counter type γ and the types of the condition variables x and y (cf. $\mathcal{C}'$). The effects of the conditional is the union of the effects of both branches. The final environment is a join of the final environments of s_1 and s_2. The join operation $\Gamma_2' \sqcup \Gamma_2''$ generates additional constraints to be included in the final constraints $\mathcal{C}$.

ST-CALL

$$\frac{\begin{array}{c}\mathcal{S} = \text{constraints}\,(M) \qquad \mathcal{G} = \text{effects}\,(M) \qquad var_1, .., var_n = \text{params}\,(M) \\ \mathcal{C}' = \mathcal{S}[\textbf{this} \mapsto \Gamma_1(x), var_1 \mapsto \Gamma_1(x_1), .., var_n \mapsto \Gamma_1(x_n), \textbf{ret} \mapsto \alpha^\dagger] \\ \mathcal{C} = \mathcal{C}' \cup \{\Gamma_1(x) \leq \alpha^\dagger, \gamma \leq \alpha^\dagger\} \cup \bigsqcap_{\gamma} \mathcal{G} \qquad \Gamma_2 = \Gamma_1[var \mapsto \alpha^\dagger]\end{array}}{\gamma \vdash var = x.M(x_1, .., x_n) : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, (\{var\}, \mathcal{G})}$$

ST-IF

$$\frac{\begin{array}{c}\beta^\dagger \vdash s_1 : \Gamma_1 \Rightarrow \Gamma_2', \mathcal{C}_1, (\mathcal{L}_1, \mathcal{G}_1) \\ \beta^\dagger \vdash s_2 : \Gamma_1 \Rightarrow \Gamma_2'', \mathcal{C}_2, (\mathcal{L}_2, \mathcal{G}_2) \qquad \mathcal{C}' = \{\gamma \leq \beta^\dagger, \Gamma_1(x) \leq \beta^\dagger, \Gamma_1(y) \leq \beta^\dagger\} \\ \Gamma_2, \mathcal{C}'' = \Gamma_2' \sqcup \Gamma_2'' \qquad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}' \cup \mathcal{C}'' \qquad \mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2 \qquad \mathcal{G} = \mathcal{G}_1 \cup \mathcal{G}_2\end{array}}{\gamma \vdash \textbf{if}\,[sl, \mathcal{L}, \mathcal{G}](x == y)\{s_1\}\{s_2\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, (\mathcal{L}, \mathcal{G})}$$

ST-CXCAST

$$\frac{\beta^\dagger \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}', (\mathcal{L}, \mathcal{G}) \qquad \mathcal{C} = \{a \leq \beta^\dagger, \gamma \leq a'\} \cup \mathcal{C}' \qquad a' \lesssim a}{\gamma \vdash (a' \Rrightarrow a)\{s\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, (\mathcal{L}, \{a'\})}$$

Figure 9 Statement typing: branching, method calls and context casts, $\boxed{\gamma \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}}$.

▶ **Definition 6** (Join of typing environments)**.** An environment Γ together with a constraint set $\mathcal{C}$ form the join of two environments Γ_1 and Γ_2, written $\Gamma, \mathcal{C} = \Gamma_1 \sqcup \Gamma_2$, iff

$$\begin{aligned}\mathcal{C} &= \{\Gamma_1(x) \leq \alpha_x \mid x \in \text{dom}(\Gamma_1)\} \cup \{\Gamma_2(x) \leq \alpha_x \mid x \in \text{dom}(\Gamma_2)\} \\ \Gamma &= \{x \mapsto \alpha_x \mid x \in \text{dom}(\Gamma_1) \cup \text{dom}(\Gamma_2)\}\end{aligned}$$

where $\{\alpha_x \mid x \in \text{dom}(\Gamma_1) \cup \text{dom}(\Gamma_2)\}$ is a set of fresh type variables, one for each variable in the domain of Γ_1 and Γ_2.

Rule ST-WHILE (not shown) works similarly to rule ST-IF. Rule ST-CXCAST deals with the cast between program counters: The program counter type of the cast context needs to subsume the outer type a' given in the cast. The outer type also defines the global effect of the statement. The body of the cast is typed under a fresh program counter type variable $\beta^\dagger$. The constraints require that $\beta^\dagger$ subsumes the inner type a. Together with the typing rules ST-PUTFIELD and ST-CALL, this restriction guarantees that the global effect $\mathcal{G}$ of the body is subsumed by $\{a\}$. Also, a' has to be castable to a.

The rule for sequences (not shown) is unsurprising: it combines constraints and effects of its components, as expected.

5 Dynamics

To enforce non-interference in dynamically checked code, LJGS' dynamics propagate and check run-time security labels. Before explaining the full operational semantics, we first discuss the interpretation of security labels and their operations.

5.1 Security Labels

Security labels (see Figure 10) are the mirror image of security types: a security label is either the static label, S, a dynamic label, D(A), carrying a security level or, the *public label*, •. The level of a static label is absent at run-time because it has been checked by the type

$$
\begin{array}{rcl}
\varsigma, \ell, g & ::= & \mathrm{S} \mid \mathrm{D}(A) \mid \bullet \\
s & ::= & \ldots \mid \mathbf{done} \\
E & ::= & \langle\rangle \mid E;\ s \mid \mathbf{cx}\,[\mathcal{L}, \mathcal{G}, \mathcal{R}, \ell]\{E\} \\
 & \mid & \mathbf{cx}\,[a/\ell/g \Rrightarrow a'/\ell/g']\{E\} \\
 & \mid & \mathbf{call}\,[M, var, x, y_1, .., y_n, L]\{E\} \\
rs & ::= & E(s)
\end{array}
\qquad
\begin{array}{rcl}
v & ::= & rv[\varsigma] \\
rv & ::= & oref \mid N \\
L & ::= & \varnothing \mid L \oplus x \mapsto v \\
 & \mid & L \oplus x \mapsto \mathbf{null} \\
\mu & ::= & \varnothing \mid \mu \oplus oref \mapsto obj
\end{array}
$$

Figure 10 Dynamic domains and run-time statements.

system already. The dynamic subset of security labels form a lattice based on the attached security levels. The entire set of labels forms a lower semi-lattice with $\bullet$ as bottom element, analogously to the semi-lattice of security types.

During the execution of an LJGS program, every value carries a *value label* ς. To track implicit flows, the semantics maintains two labels for each execution context, a *local program counter label*, ℓ, and a *global program counter label*, g. For value labels, this security level indicates the current run-time confidentiality of a labeled value. A program counter label approximates the confidentiality of information that implicitly flows into the current program counter. Thus, the dynamic label on a program counter gives a lower bound on the side effects.

When information flows into a value or program counter from multiple sources, the dynamic semantics employs a partial join operation on the labels involved. The judgment $\varsigma := \varsigma_1 \sqcup \varsigma_2$ determines the label ς that joins ς_1 and ς_2.

$$
\frac{}{\mathrm{S} := \mathrm{S} \sqcup \mathrm{S}}
\qquad
\frac{}{\mathrm{D}(A_1 \sqcup A_2) := \mathrm{D}(A_1) \sqcup \mathrm{D}(A_2)}
\qquad
\frac{}{\varsigma := \bullet \sqcup \varsigma}
\qquad
\frac{}{\varsigma := \varsigma \sqcup \bullet}
$$

Joining static labels is trivial. Dynamic labels are joined by joining the security levels they contain. The public label is a neutral element for the join operation. Joining static and dynamic labels is undefined, as the security level of a static labeled entity cannot be recovered at run-time. A well typed LJGS program only performs well-defined joins.

5.2 Configurations

The execution of a method manipulates *configurations* $rs/L/\mu$ of run-time statements rs, *stack frames* L, and *heaps* μ. Run-time statements, defined in Figure 10, consist of execution contexts E applied to source statements s. For technical reasons, the statements previously defined in Section 3 need to be extended with the marker statement **done**. An execution context is a hole $\langle\rangle$, a sequence of an execution context and a source statement, a *run-time security context* $\mathbf{cx}\,[\mathcal{L}, \mathcal{G}, \mathcal{R}, \ell]\{E\}$, a *cast context* $\mathbf{cx}\,[a/\ell/g \Rrightarrow a'/\ell'/g']\{E\}$, or a *calling context* $\mathbf{call}\,[M, var, x, x_1, .., x_n, L]\{E\}$. The run-time statement $\langle\mathbf{done}\rangle$ signals that execution of the current context is complete. A sequence context focuses the first component of a statement sequence. A run-time security context remembers metadata that is needed to track the dynamic implicit flows that arise from branching statements and virtual method calls: the program counter label ℓ that is active during its execution, and local and global effects, $\mathcal{L}$, $\mathcal{G}$, that were analyzed for its body. It also remembers a set of *field references* $\mathcal{R}$ which we explain in Sections 5.3 and 5.4. A cast context stores the old and new program counter types and labels during the execution of a block subject to a context cast. A calling context stores the method name and stack frame of the callee and the return variable of the caller during a method call. It also contains the method call receiver x and the call arguments $x_1, \ldots, x_n$ for technical reasons.

A stack frame is a finite map from local variables to *values.* Stack frames have a fixed size and bind exactly the local variables that occur in the body of their method. Uninitialized variables are bound to **null**.

A value consists of a *raw value rv* and value label ς. A raw value is either a *number* $N \in \mathbb{N}$ or an *object reference*, *oref.* A heap is a finite map from *references* to *objects.* An object consists of its run-time class C and the values of its fields.

5.3 Effect Analysis Support for Dynamic IFC

Information flow control in the dynamic fragment of LJGS copes with implicit flows by relying on statically analyzed information about write effects in untaken control flow branches.[6] As in Russo and Sabelfeld's work [23] the basic principle is to upgrade the labels of dynamic variables and fields if they may be updated in the untaken branches.

For upgrades of local variables the typing rules generate sufficient information with the local effect $\mathcal{L}$. For object oriented languages like LJGS, obtaining precise information about heap write effects is more involved. Including such an analysis in LJGS would go beyond the scope of this paper. Instead, we rely on an external static analysis for write effects that provides (abstract) references to potentially updated fields in branching statements and method calls. In the semantics, we encapsulate the analysis result in the meta-function call $\mathcal{R} = \text{updaterefs}\,(sl, L, \mu)$. The function updaterefs takes as arguments the location of a branching statement (or the name of a method), a stack frame, and a heap, and returns a set $\mathcal{R}$ of (concrete) field references *oref.F*. The result $\mathcal{R}$ contains references to all fields that may be updated by the statement labelled sl, the concretization of the externally analyzed write effect for L and μ. The technical report gives a precise specification of updaterefs.

There are a number of interprocedural points-to analyses for languages with heap references [18, 28] from which a write effect analysis suitable for implementing updaterefs can be derived. We limit the static analysis to write effects for simplicity; it should also be possible to directly incorporate points-to information into the dynamic IFC policy, as illustrated by Moore and Chong [20], to gain precision.

5.4 Reduction rules

The judgment $\ell;\ g \vdash rs/L/\,\mu \longrightarrow rs'/L'/\,\mu'$ defines the small step reduction of a configuration under the local program counter label ℓ and global program counter label g. Apart from the rules for casts, reduction mostly follows the principles of other dynamic IFC systems [3, 23]. Figures 11 and 12 show some of the rules. The technical report contains the full semantics.

Figure 11 shows the rules for evaluating expressions and for updating local variables. The judgment $\vdash e/L/\,\mu \Downarrow v/\,\mu'$ evaluates expression e to a result and an updated heap. Rule STEP-UPD-PLUS illustrates evaluation. It reads the operands from the stack frame L, and returns a raw value, here $N + N'$, that has an updated security label attached. The updated label is the join of the operands' labels. Allocations, covered by rule STEPUPD-NEW, extend the heap with a fresh reference which has the public label. Other expressions that are not shown work similarly. A cast expression converts the value label of its subject according to the label conversion judgment $a \Rrightarrow a' \vdash \varsigma \Rrightarrow \varsigma'$, also defined in Figure 11. Label conversion

[6] IFC techniques where the label propagation relies on static analysis are typically called *hybrid IFC*. Given our setting, we still refer to these approaches as dynamic IFC because they are based on run-time propagation of security labels and because LJGS employs such an approach in the dynamically typed fragments of programs.

$$\textsc{stepupd-plus}\quad \frac{N[\varsigma_1] = L[x] \qquad N'[\varsigma_2] = L[y] \qquad \varsigma := \varsigma_1 \sqcup \varsigma_2}{\vdash x + y / L / \mu \Downarrow N + N'[\varsigma] / \mu}$$

$$\textsc{stepupd-new}\quad \frac{v_1 = L[x_1] \quad .. \quad v_n = L[x_n] \qquad \mathit{oref} = \mathit{fresh} \qquad v' = \mathit{oref}[\bullet] \qquad \mu' = \mu \oplus \mathit{oref} \mapsto \{C, v_1 \mathinner{..} v_n\}}{\vdash \mathbf{new}\, C(x_1, \mathinner{..}, x_n) / L / \mu \Downarrow v' / \mu'}$$

$$\textsc{stepupd-cast}\quad \frac{rv[\varsigma] = L[x] \qquad a' \Rrightarrow a \vdash \varsigma \Rrightarrow \varsigma'}{\vdash (a \Lleftarrow a')x / L / \mu \Downarrow rv[\varsigma'] / \mu}$$

$$\textsc{estep-local}\quad \frac{\vdash e / L / \mu \Downarrow rv[\varsigma] / \mu' \qquad \varsigma' := \varsigma \sqcup \ell \qquad L' = L[\mathit{var} \mapsto rv[\varsigma']]}{\ell;\ g \vdash \langle \mathit{var} = e \rangle / L / \mu \longrightarrow \langle \mathbf{done} \rangle / L' / \mu'}$$

$$\frac{}{a \Rrightarrow a \vdash \varsigma \Rrightarrow \varsigma} \qquad \frac{}{A \Rrightarrow \star \vdash \varsigma \Rrightarrow \mathrm{D}(A)} \qquad \frac{A \sqsubseteq A'}{\star \Rrightarrow A' \vdash \mathrm{D}(A) \Rrightarrow \mathrm{S}} \qquad \frac{}{\bullet \Rrightarrow A \vdash \varsigma \Rrightarrow \mathrm{S}}$$

$$\frac{}{\bullet \Rrightarrow \star \vdash \varsigma \Rrightarrow \mathrm{D}(\bot)} \qquad \frac{}{\bot \Rrightarrow \bullet \vdash \mathrm{S} \Rrightarrow \bullet} \qquad \frac{}{\star \Rrightarrow \bullet \vdash \mathrm{D}(\bot) \Rrightarrow \bullet}$$

Figure 11 Local updates (excerpt) $\boxed{\vdash e / L / \mu \Downarrow v / \mu'}$ and label conversion $\boxed{a' \Rrightarrow a \vdash \varsigma' \Rrightarrow \varsigma}$.

changes a label ς that corresponds to a cast's source type a to a label ς' corresponding to the destination type a'. A trivial cast results in a trivial label conversion. The conversion from a static type to the dynamic type creates a dynamic label that contains the source type as security level. A static-to-dynamic conversion always succeeds in a well-typed LJGS program. A conversion from dynamic to a static type A' returns the static label, if the dynamic source label contains a security level subsumed by A. Otherwise, the static-to-dynamic conversion fails. The public type may be converted to a dynamic type and a low-security label or any static type. Converting to the public type requires either a static source type of bottom or a dynamic bottom label. Finally, rule ESTEP-LOCAL stores the result of an evaluation in a local variable after joining its label with the current program counter label.

Figure 12 shows illustrative cases of reductions that enter and leave contexts. Rule ESTEP-IF-TRUE covers if-statements where the branch condition is satisfied. Selecting a branch potentially creates an implicit flow from the condition to the execution context of the branch. Thus, the reduction results in a run-time security context that stores a program counter label ℓ'' which includes the value labels of the operands and the label of the program counter in the outer context. The run-time context also stores the effect annotations of the conditional and the field reference set $\mathcal{R}$ that points to the heap locations that are potentially modified during the execution the untaken branch s_2. The rules for failing conditions, while loops and method calls (not shown) are similar. Method calls additionally create a calling context **call** $[M, \mathit{var}, x, y_1, \mathinner{..}, y_n, L]\{E\}$ that initializes new stack frames and copies the method result into the caller's stack frame in a straightforward way.

Rule ESTEP-CX-DONE exits the run-time context and, to counter illegal implicit flows, performs a preemptive upgrade of local variables and fields mentioned in the context. The meta functions $\mathbf{upgr}(\mu, \mathcal{R}, \mathcal{G}, \ell)$ and $\mathbf{upgr}(L, \mathcal{L}, \mathcal{G}, \ell)$ perform these upgrades for all field references in $\mathcal{R}$ and all (dynamic) variables in $\mathcal{L}$, respectively. The functions use the program counter label for upgrading, if it is dynamic. But, due to context casts, upgrades may also

ESTEP-IF-TRUE

$$\frac{rv[\varsigma_1] = L[x] \qquad rv[\varsigma_2] = L[y] \qquad \varsigma' := \varsigma_1 \sqcup \varsigma_2 \qquad \ell'' := \ell \sqcup \varsigma' \qquad \mathcal{R} = \text{updaterefs}\,(sl, L, \mu)}{\ell;\ g \vdash \langle \textbf{if}\ [sl, \mathcal{L}, \mathcal{G}](x == y)\{s_1\}\{s_2\}\rangle / L /\ \mu \longrightarrow \textbf{cx}[\mathcal{L}, \mathcal{G}, \mathcal{R}, \ell'']\{\langle s_1 \rangle\}/L/\ \mu}$$

ESTEP-CX-DONE

$$\frac{\mu' = \textbf{upgr}(\mu, \mathcal{R}, \mathcal{G}, \ell') \qquad L' = \textbf{upgr}(L, \mathcal{L}, \mathcal{G}, \ell')}{\ell;\ g \vdash \textbf{cx}[\mathcal{L}, \mathcal{G}, \mathcal{R}, \ell']\{\langle \textbf{done} \rangle\}/L/\ \mu \longrightarrow \langle \textbf{done} \rangle / L' /\ \mu'}$$

ESTEP-CXCAST

$$\frac{a \Rrightarrow a' \vdash \ell \Rrightarrow \ell' \qquad a \Rrightarrow a' \vdash g \Rrightarrow g'}{\ell;\ g \vdash \langle (a \Rrightarrow a')\{s\} \rangle / L /\ \mu \longrightarrow \textbf{cx}[a/\ell/g \Rrightarrow a'/\ell'/g']\{\langle s \rangle\}/L/\ \mu}$$

Figure 12 Reduction: entering and leaving contexts (excerpt) $\boxed{\ell;\ g \vdash rs/L/\ \mu \longrightarrow rs'/L'/\ \mu'}$.

be necessary in static contexts where the program counter label ℓ carries no information. For example, consider the following statement that contains a context cast inside of a conditional:

```
if(this.high) {(HIGH ⇛ ⋆) { this.dyn = 42; }}
```

Here, the type of field `high` is the static level `HIGH` and that of `dyn` is dynamic. According to the IFC principle outlined above, the semantics should upgrade `this.dyn` after executing the statement, as a dynamic field is conditionally updated with a high-security program counter. In those situations, the upgrade functions use the greatest lower bound of the global effects $\mathcal{G}$ for upgrading. This choice is safe because the source type of context casts are always reflected in the global effects (cf. typing rule ST-CXCAST). The technical report gives a complete definition of the **upgr** functions.

Context casts, covered by rule ESTEP-CXCAST, determine the local and global program counter labels for executing their body with the same label conversion judgment used for value casts. A cast execution context stores the converted and original types and labels. Reduction under a run-time security context, covered by rule ESTEP-CX-STEP, uses the stored, modified program counter label. The global program counter label is joined with the stored label such that global effects also respect the augmented context. When a run-time security context is completely executed, it is discarded (not shown). Cast contexts (not shown) work similarly, but use the stored converted labels to reduce their bodies. Sequence execution contexts, also not shown, completely execute the first context before moving on to the second.

6 Execution Model

The dynamics of LJGS attach security labels to static and dynamic values alike. Fortunately, LJGS is designed such that static labels can be erased in a realistic implementation. In this section, we sketch a compilation strategy from Java with LJGS security annotations to plain Java. The actual implementation of a compiler to bytecode is ongoing work.

Figure 13 shows an LJGS class with a dynamic field `F`, a statically `LOW` field `G` and a method `m` that includes typecasts. LJGS specific annotations and casts are coloured blue. The Figure also shows, in red, the Java code that an LJGS compiler would produce according to our execution model. The blue parts would be removed from the the compilation result. First, the required datastructures are brought into scope: Line 5 retrieves a `LocalMap locals`

```
1  class C { int F[⋆]; int G[LOW];
2
3   void m(int y) where { y ∼ ⋆ }
4                 and {⋆, LOW} {
5    LocalMap locals = getCall();
6    ObjectMap objects = getObjects();
7    PcStack lPC = new PcStack();
8    PcStack gPC = getGlobalPC();
9    int g = this.G;
10   int x = (⋆ ⇐ HIGH) g;
11   locals.put("x", Levels.HIGH);
12   x = x + y;
13   locals.put(
14     "x", join(locals.get("x"),
15               locals.get("y")));
16   lPC.push(locals.get("x"));
17   gPC.push(locals.get("x"));
18   if [this.F] (x == 5) {
19     this.F = 42;
20     objects.put(this, "F", gPC.get());
21   }
22   objects.put(this, "F",
23    join(objects.get(this, "F"),
24         gPC.get()));
25   lPC.pop(); gPC.pop();
26   int y = this.F
27   locals.put("y", objects.get(this, "F"));
28   cast(locals.get("y"), Level.LOW);
29   this.G = (LOW ⇐ ⋆) y;
30  }
31 }
```

Figure 13 An LJGS class compiled to plain Java code.

that maps local variables to security levels. Each executing method call possesses its own `LocalMap`. The caller initializes the `LocalMap` with the security levels of the dynamic method arguments and publishes it for the callee using the method `setCall` as illustrated with the following code fragment:

```
int x = ... /* some dynamic variable */
LocalMap localsForM = new LocalMap();
localsForM.put("x", locals.get("x"));
setCall(localsForM);
m(x);
```

The globally accessible `ObjectMap objects` (line 6) is a weak map from objects and field names to security levels which stores the dynamic labels of all non-null dynamic fields.

Lines 7 and 8 retrieve the `PcStack` objects to track the levels of dynamic local and global program counters. A `PcStack` is a stack of security levels that stores the level of each dynamic branch condition or method call receiver. The join of all security levels on a stack, calculated by `PcStack.get()`, yields the level of the current, dynamic program counter. The local program counter stack is freshly initialized for the method while the global program counter stack is available through the static method `getGlobalPC()`.

Line 9 is the first statement taken from the original LJGS program. As it is a static update, no instrumentation is needed. Lines 10 and 11 perform the cast from `HIGH` to $\star$. As casts from static to dynamic never fail, the value of `g` is copied to `x`. Line 11 stores the source level mentioned in the cast for `x` in `locals`. Similarly, line 14 updates the local map for `x` with the join of the labels of `x` and `y`, using `locals` and the static method `join`.

Line 18 starts a context with a dynamic program counter. Thus, the dynamic label of the tested variable `x` gets pushed to the local and global `PcStacks`. The field update in line 19 with a low-security constant requires an update to the object map with the label of the global pc (line 20). Line 22 forms the join point for the conditional. To protect against implicit flows, the label of field `F` is joined with the level of the global program counter. Afterwards, line 25 pops the program counter stacks. The cast of the dynamic variable to the static security level `LOW` compiles to lines 28 and 29. The static method `cast` checks if `y`'s label can be converted to `LOW`. If not, as is the case in this example, the program is aborted. Otherwise line 29 performs the field update no further dynamic security tracking of `y`'s value.

The compilation of LJGS can rely on the typing derivation of `m` to determine whether code is dynamic or static and thus whether label tracking code should be emitted: For example, line 9 in Figure 13 does not require tracking code as it is a static update with a public program counter, whereas line 10 is a dynamic update that needs to be tracked.

Polymorphic methods, like method `max` of Figure 1, can be called with dynamic or static arguments. To avoid overhead in the static case, a compilation procedure can generate different versions of `max`, say `max_STATIC_STATIC` and `max_DYN_DYN`, for static and dynamic calls, respectively. The Java method `max_STATIC_STATIC` would not contain any tracking code whereas `max_DYN_DYN` would track all updates and program counters. The corresponding results are not shown here for lack of space but included in the technical report. It is not necessary to generate versions for other combinations of parameters, like `max_STATIC_DYN`, as they are ruled out by the method constraints.

7 Unanalyzable Code

LJGS, as presented in the previous sections, relies on two kinds of static analysis: A type analysis checks the typing rules of Section 4. It requires all classes to be annotated with field types and method signatures, and checks all method bodies for compliance with their signature. Additionally, a write effect analysis supports the dynamic IFC (cf. Section 5.3).

From a practical standpoint, a major use case for gradual typing is the integration of legacy code that is hard to type and analyze statically. In addition, one might expect that dynamically typed code could take advantage of highly dynamic language features like reflection. This section explains how to extend LJGS to accommodate for code that cannot be easily annotated or analyzed. We require, however, that it is possible to *instrument* this unanalyzable code, either by source- or bytecode-transformation or by integrating a corresponding monitor to the virtual machine. We thus do not consider legacy code that is impractical to instrument, like precompiled C-libraries.

7.1 Default, Dynamic Type Annotations

The type-checking of unanalyzeable code can be avoided by assuming dynamic annotations by default and using dynamic IFC during its execution. Consider the following example where `LegacyClass` represents a class that should not be subject to (security-) type-checking.

```
1   class LegacyClass { int legacyField; int legacyMethod(int x, int y) {...}}
2   class LJGSClass {void m() {z = reflectiveCall("any" + "Method", x, y); }}
```

The default annotation for the field `legacyField` would be $\star$ and the default constraints and effect for method `legacyMethod` would be $\{\star \leq \mathbf{ret}, x \leq \star, y \leq \star\}$ and $\{\star\}$, respectively. As long as `legacyMethod` does not use LJGS specific features like casts and only accesses other legacy classes, the method's body will comply with the default signature; type-checking would only generate trivial constraints that classify all entities as dynamic.

Care has to be taken in the presence of abstraction-breaking features like reflection which could allow legacy methods to access statically typed fields and methods of LJGS classes that derive from legacy classes. If these features are required, the run-time enforcement needs to be extended to dynamically block access to non-legacy classes.

A reflective call, as illustrated in line 2, has to be treated in the same way as legacy code. Thus, the run-time enforcement needs to check dynamically if the called method, here `anyMethod`, complies to a default signature.

7.2 Avoiding the Analysis of Write Effects

If a precise write effect analysis is infeasible for legacy code, a more conservative approximation can be used, for example, by assuming that *all* fields of all exposed classes of a legacy Java library are updated when calling a legacy method.

Another possibility is to use a *purely dynamic* enforcement, like Austin and Flanagan's non-sensitive-upgrade policy (NSU, [3]). However, a purely dynamic approach may report false positives that would pose no problems for flow-sensitive static typing [23] whereas a hybrid approach, as the one taken by LJGS, is strictly more flexible.

Instead of falling back to purely dynamic IFC, it is possible to assert a particular write effect for legacy method calls and reflective calls and check compliance dynamically using a dynamic effect analysis, like access permission contracts [15].

8 Correctness

To guarantee that the statics and dynamics presented in the previous sections enforce security according to our attacker model, we need to prove termination insensitive non-interference. As the dynamics only check the security labels of dynamically typed values, non-interference also relies on the soundness of the type system which guarantees that statically typed data is not responsible for program crashes caused by security violations. We always implicitly assume that an LJGS program is well-formed according to the standard typing principles of Java. In particular we rely on the fact that no variables or object fields are accessed uninitialized. Also, we implicitly assume that the class hierarchy in an LJGS program is well-typed, according to Definition 5. Proof sketches of the theorems stated in this section can be found in the technical report.

8.1 Soundness of the type system

Simple typing judgments for values and annotations connect static and dynamic domains.

▶ **Definition 7** (Typing of Dynamic Domains). A **security label** ς **has type** a, written $\vdash \varsigma : a$, if either (i) $\varsigma = \bullet$, (ii) $\varsigma = \mathrm{D}(A)$ and $a = \star$, or (iii) $\varsigma = \mathrm{S}$ and $a = A$. A **security label is typed by a type variable** α under constraints $\mathcal{C}$, written $\mathcal{C} \vdash \varsigma : \alpha$, if there exist a and θ such that $\theta \models (\mathcal{C} \cup \{\alpha \leq a\})$ and $\vdash \varsigma : a$. A **stack frame** L is well-typed under constraints $\mathcal{C}$ and environment Γ, written $\Gamma, \mathcal{C} \vdash L$ if for all bindings $x \mapsto rv[\varsigma]$ in L it holds that $\mathcal{C} \vdash \varsigma : \Gamma(x)$. This judgment essentially checks that dynamically typed variables have either dynamic or public labels. A **heap** μ is well-typed, written $\vdash \mu$, if for all field values it contains, the value label has the type of the corresponding field declaration.

The type soundness and non-inference lemmas require a typing judgment for execution contexts, $\gamma, \ell, g \vdash E : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}|\ell', g'$. The judgment describes the typing environment of the execution context, and, given its program counter labels ℓ, g, it defines the program counter labels ℓ', g' that are active at its hole. For example, starting with labels ℓ, g, the context $\mathbf{cx}\,[\mathcal{L}, \mathcal{G}, \mathcal{R}, \ell']\{\langle\rangle\}$ has the labels ℓ' and $\ell' \sqcup g$ active at its hole. The constraints, effect and environments that make up a context typing are the same as for statement typing. The technical report contains the corresponding rules. They are unsurprising; the premises for each context (e.g. **cx**) are similar to those of the statements that enter the context (**if**).

A typed configuration $E(s)/L/\mu$ combines the typings of its individual components. Thus we write $\gamma, \ell, g \vdash E(s)/L/\mu : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}|\ell', g'$ if (i) $\Gamma_1, \mathcal{C} \vdash L$ (ii) $\vdash \mu$ and $\gamma, \ell, g \vdash E : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}|\ell', g'$, (iii) $\gamma \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}$ and $\mathcal{C} \vdash \ell : \gamma$ (iv) $\ell \leq g$. As usual for a gradually typed system, a well typed LJGS program guarantees a refined progress property. While (security related) run-time errors in static code are ruled out, a program may run into a *dynamically stuck* configuration where dynamic code goes wrong.

▶ **Definition 8** (Dynamically stuck). A well-typed configuration $\gamma, \ell, g \vdash E(s)/L/\mu : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}|\ell', g'$ is *dynamically stuck* iff it attempts either: (i) an insecure value cast from dynamic to static, $s = (var = (a \Leftarrow \star)x)$, $a \in \{\bullet, A\}$, $L[x] = rv[\mathrm{D}(A')]$, and $a \not\leq A$, or (ii) an insecure context cast from dynamic to static $s = ((\star \Rrightarrow a)\{s'\})$, $\ell' = \mathrm{D}(PC)$, $a \in \{\bullet, PC\}$and $PC \not\sqsubseteq A$

The progress result for LJGS states that well typed programs that are not able to make a reduction step are either **done**, or dynamically stuck. The preservation result is standard: a well typed LJGS configuration that can be reduced yields another well-typed configuration with possibly more permissible constraints and effects. The precise definitions for progress and preservation are given in the technical report.

8.2 Non-Interference

The non-interference theorem for LJGS states that methods run under *low-equivalent environments*, that is, environments that an attacker cannot distinguish, produce *low-equivalent results*. First we define the notion of *low-equivalence* and subsequently state the non-interference theorem. In the following, we assume that *low* is the upper bound of security levels that an attacker can observe. We refer to a security level A as *high* if $A \not\sqsubseteq low$.

▶ **Definition 9** (Low-equivalent values, objects, heaps and stack-frames). Let B be a (partial) bijective mapping on heap references (*orefs*). **Two values** $rv_1[\varsigma], rv_2[\varsigma]$ are equivalent under B, written $rv_1[\varsigma] =_B rv_2[\varsigma]$ iff either $rv_{1/2}$ are equal integers, or $rv_1 \in \mathrm{dom}\,(B)$ and $B(rv_1) = rv_2$. **Two objects** obj_1, obj_2 are low-equivalent under B, written $obj_1 =_{B,low} obj_2$, if for all fields F where $\mathrm{getfield}\,(F, obj_1) \neq_B \mathrm{getfield}\,(F, obj_2)$ it holds that either (i) $\mathrm{fsec}\,(F) = A$ and $A \not\sqsubseteq low$, or (ii) $\mathrm{fsec}\,(F) = \star$, $\mathrm{getfield}\,(F, obj_1) = rv_1[\mathrm{D}(A_1)]$, $\mathrm{getfield}\,(F, obj_2) = rv_1[\mathrm{D}(A_2)]$, and $A_1 \not\sqsubseteq low$ and $A_2 \not\sqsubseteq low$. **Two heaps** μ_1, μ_2 are low-equivalent, written $\mu_1 \equiv_{B,low} \mu_2$ if (i) $\mathrm{dom}\,(\mu_1) \supseteq \mathrm{dom}\,(B)$ and $\mathrm{dom}\,(\mu_2) \supseteq \mathrm{dom}\,(B^{-1})$, and (ii) the objects stored at the references listed in B are low-equivalent with respect to B. **Two environments** L_1, L_2 are low-equivalent with respect to environment Γ, solution θ, and bijection B, written $L_1 \equiv_{\Gamma,\theta,B,low} L_2$ if $\mathrm{dom}\,(L_1) = \mathrm{dom}\,(L_2)$, $L_1[\textbf{this}] = L_2[\textbf{this}]$, and for all $var \in \mathrm{dom}\,(L_1)$ either (i) $L_1[var] = rv_1[\varsigma_1], L2[var] = rv2[va2]]$, and $rv_1 =_B rv_2$, (ii) $\theta(\Gamma(var)) = A$, and $A \not\sqsubseteq low$, or (iii) $\theta(\Gamma(var)) = \star$ and $\varsigma_1 = \mathrm{D}(A_1)$, $\varsigma_2 = \mathrm{D}(A_2)$, and $A_1 \not\sqsubseteq low$ and $A_2 \not\sqsubseteq low$

▶ **Theorem 10** (Non-interference). *Let $E(s)/L_1/\mu_1$, $E(s)/L_2/\mu_2$ be two configurations with typings $\gamma, \ell, g \vdash E(s)/L_i/\mu_i : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}|\ell', g'$, ($i \in \{1,2\}$) and solution $\theta \models \mathcal{C}$. Let B be a bijection such that $L_1 \equiv_{\Gamma_1,\theta,B,low} L_2$ and $\mu_1 \equiv_{B,low} \mu_2$. Given two executions $\ell;\ g \vdash E(s)/L_i/\mu_i \longrightarrow^* \langle\textbf{done}\rangle/L_i'/\mu_i'$ there exists a bijection $B' \supseteq B$ such that $L_1' \equiv_{\Gamma_2,\theta,B',low} L_2'$ and $\mu_1' \equiv_{B',low} \mu_2'$.*

9 Related Work

There is a large body of prior work on security type systems that ultimately goes back to Denning and Denning's classic paper on information flow security [11]. We focus on discussing works on security type systems and dynamic security enforcement designed for "main-stream" programming languages and defer the reader to the overview articles of Sabelfeld and Myers [24] and Hedin and Sabelfeld [16] for other aspects of language based security.

FlowCaML [22] is an ML dialect supporting static polymorphic security types with security constraints similar to those of LJGS. FlowCaML additionally supports higher-order types and complete type inference. Sun, Banerjee and Naumann describe a modular polymorphic type system for a Java-like, object oriented language [30]. The polymorphic method signatures are comparable to the static fragment of LJGS, but they do not specify a constraint-based typechecking algorithm. Instead, they apply a standard algorithm to all instances of signatures. Their system additionally supports class definitions with security type parameters and full type inference. Both approaches seem compatible with LJGS and we plan to investigate how they could be adapted for gradual security typing in future work. Barthe et al describe an information flow type system for Java Bytecode and develop a corresponding certified type checker [7]. Their system supports Objects, virtual, monomorphic methods, exceptions and arrays. JIF [21] is an extension to Java with static security types and first-class dynamic security-labels. In JIF, security types may depend on dynamic labels and the interaction of labels and types is verified statically during type checking. In contrast, LJGS does not restrict dynamically labeled values statically but enforces non-interference at run-time. Also, while JIF's dynamic labels may be used to implement some form of ad-hoc dynamic IFC, LJGS aims for the integration of principled dynamic IFC techniques.

LJGS' run-time security enforcement for values with dynamic labels is an adaption of Russo and Sabelfeld's technique for hybrid information flow control [23]. Chandra and Franz [9] present an implementation of a hybrid IFC framework for Java Bytecode that is based on the same principles and closely resembles LJGS' enforcement for dynamic fragments. The updaterefs function that supports dynamic IFC in LJGS can be implemented with off-the-shelf points-to analysis like that proposed by Khedker et al [18]. We refer the interested reader to the survey paper of Smaragdakis and Balatsouras [28] for further related work on static points-to analysis. Moore and Chong [20] studied the applicability of static points-to analyses to improve the efficiency of hybrid IFC. Using the points-to information in the hybrid monitor allows their system to infer when heap values do not need tracking anymore. Although their ideas seem compatible with LJGS, our hybrid enforcement settles for an analysis of simple write effects and we rely on explicit static typing for optimization.

Austin and Flanagan propose a series of sound, purely dynamic IFC techniques, the most basic being the no-sensitive-upgrade policy, that do not rely on prior static analysis [3, 4, 5]. These approaches are also compatible with LJGS' type system.

Recently, Bedford et al proposed a type system for a simple imperative core language that also includes types for entities with statically unknown security levels [8]. Their work focuses on inferring program points for run-time instrumentation based on the typing results. In contrast, LJGS' focus is on giving the programmer explicit control over the boundaries between static and dynamic checking.

The original work on gradual typing [32, 19, 27, 34] focuses on simple types with extensions like refinement predicates, polymorphism [1], and union types [33]. More recently, researchers started to gradualize type systems that check properties unrelated to the structure of values, like type annotations [14], ownership [25], typestate [35], and session types [31]. Gradual security type systems also fall into this category. Disney and Flanagan study gradual security types for a pure lambda calculus [12] and we describe MLGS, a system for a calculus with ML style references, in our prior work [13]. Compared with LJGS treatment of object fields, MLGS treats mutable references in a more liberal way because it admits casts between reference types with static and dynamic content. However, this liberality comes at the cost of requiring pervasive run-time labelling even for static values and it blurs the separation of static and dynamic code, which, although sound, runs contrary the execution model and design goal of LJGS.

10 Conclusion and Future Work

LJGS is a sequential Java core calculus with gradual security typing. The calculus strictly separates statically verified and dynamically checked code which enables running statically checked code without run-time security labels. Methods may have polymorphic security signatures that accept static or dynamic arguments.

There are several avenues for future work. We are currently implementing the type checking and run-time enforcement for (sequential) Java based on the principles of LJGS. A type checker is already available as an accompanying artifact to this paper but the run-time instrumentation is still work in progress. With this implementation we want to investigate the practicality of the type system for realistic applications and to evaluate different compilation- and execution strategies for dynamic code. We also want to extend the system with security type parameters for classes and methods, and type inference.

References

1 Amal Ahmed, Robert Bruce Findler, Jacob Matthews, and Philip Wadler. Blame for all. In *Proceedings of the 1st Workshop on Script to Program Evolution*, pages 1–13, Genova, Italy, 2009. ACM.

2 Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David 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 Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In Stephen Chong and David A. Naumann, editors, *PLAS*, pages 113–124, Dublin, Ireland, June 2009. ACM.

4 Thomas H. Austin and Cormac 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 Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In John Field and Michael Hicks, editors, *Proc. 39th ACM Symp. POPL*, pages 165–178, Philadelphia, USA, January 2012. ACM Press.

6 Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *CSFW*, pages 253–, Cape Breton, Nova Scotia, Canada, June 2002. IEEE Computer Society.

7 Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference Java bytecode verifier. *Mathematical Structures in Computer Science*, 23(5):1032–1081, 2013.

8 Andrew Bedford, Josée Desharnais, Théophane G. Godonou, and Nadia Tawbi. Enforcing information flow by combining static and dynamic analysis. In Jean Luc Danger, Mourad Debbabi, Jean-Yves Marion, Joaquín García-Alfaro, and A. Nur Zincir-Heywood, editors, *Foundations and Practice of Security - 6th International Symposium, FPS 2013, La Rochelle, France, October 21-22, 2013, Revised Selected Papers*, volume 8352 of *LNCS*, pages 83–101, La Rochelle, France, October 2013. Springer.

9 Deepak Chandra and Michael Franz. Fine-grained information flow analysis and enforcement in a Java virtual machine. In *23rd Annual Computer Security Applications Conference (ACSAC 2007)*, pages 463–475, Miami Beach, Florida, USA, December 2007. IEEE Computer Society.

10 Dorothy Denning. A lattice model of secure information flow. *Comm. ACM*, 19(5):236–242, 1976.

11 Dorothy Denning and Peter Denning. Certification of programs for secure information flow. *Comm. ACM*, 20(7):504–513, 1977.

12 Tim Disney and Cormac Flanagan. Gradual information flow typing. In *STOP*, 2011.

13 Luminous Fennell and Peter Thiemann. Gradual security typing with references. In Véronique Cortier and Anupam Datta, editors, *CSF*, pages 224–239, New Orleans, LA, USA, 2013. IEEE.

14 Luminous Fennell and Peter Thiemann. Gradual typing for annotated type systems. In Zhong Shao, editor, *ESOP'14*, volume 8410 of *Lecture Notes in Computer Science*, pages 47–66, Grenoble, France, April 2014. Springer.

15 Manuel Geffken and Peter Thiemann. Side effect monitoring for Java using bytecode rewriting. In Joanna Kolodziej and Bruce R. Childers, editors, *PPPJ '14*, pages 87–98, Cracow, Poland, September 2014. ACM.

16 Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In *2011 Marktoberdorf Summer School*. IOS Press, 2011.

17 Sebastian Hunt and David Sands. On flow-sensitive security types. In Simon Peyton Jones, editor, *Proc. 33rd ACM Symp. POPL*, pages 79–90, Charleston, South Carolina, USA, January 2006. ACM Press.

18 Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *ACM TOPLAS*, 30(1), 2007.

19 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In Matthias Felleisen, editor, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 3–10, Nice, France, January 2007. ACM Press.

20 Scott Moore and Stephen Chong. Static analysis for efficient hybrid information-flow control. In *CSF 2011*, pages 146–160, Cernay-la-Ville, France, June 2011. IEEE Computer Society.

21 Andrew C. Myers. JFlow: Practical mostly-static information flow control. In Alexander Aiken, editor, *Proc. 26th ACM Symp. POPL*, pages 228–241, San Antonio, Texas, USA, January 1999. ACM Press.

22 François Pottier and Vincent Simonet. Information flow inference for ML. *ACM TOPLAS*, 25(1):117–158, January 2003.

23 Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *CSF*, pages 186–199. IEEE Computer Society, 2010.

24 Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

25 Ilya Sergey and Dave Clarke. Gradual ownership types. In *21th European Symposium on Programming (ESOP 2012)*, Tallinn, Estonia, April 2012. Springer.

26 Jeremy Siek and Walid Taha. Gradual typing for objects. In Erik Ernst, editor, *21st ECOOP*, volume 4609 of *LNCS*, pages 2–27, Berlin, Germany, July 2007. Springer.

27 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.

28 Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.

29 Rok Strnisa and Matthew J. Parkinson. Lightweight Java. *Archive of Formal Proofs*, 2011, 2011.

30 Qi Sun, Anindya Banerjee, and David A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In Roberto Giacobazzi, editor, *SAS 2004*, volume 3148 of *LNCS*, pages 84–99, Verona, Italy, August 2004. Springer.

31 Peter Thiemann. Gradual typing for session types. In Emilio Tuosto and Matteo Maffeis, editors, *TGC*, volume 8902 of *LNCS*, pages 144–158, Rome, Italy, September 2014. Springer.

32 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium, DLS 2006*, pages 964–974, Portland, Oregon, USA, 2006. ACM.

33 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In Phil Wadler, editor, *Proc. 35th ACM Symp. POPL*, pages 395–406, San Francisco, CA, USA, January 2008. ACM Press.

34 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Proc. 18th ESOP*, volume 5502 of *LNCS*, pages 1–16, York, UK, March 2009. Springer.

35 Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *ECOOP*, volume 6813 of *LNCS*, pages 459–483, Lancaster, UK, 2011. Springer.

36 Stephan Arthur Zdancewic. *Programming Languages for Information Security.* PhD thesis, Cornell, Ithaca, NY, USA, 2002.