2009 22nd IEEE Computer Security Foundations Symposium

Updatable Security Views

J. Nathan Foster Benjamin C. Pierce Steve Zdancewic
University of Pennsylvania

Abstract

Security views *are a flexible and effective mechanism for controlling access to confidential information. Rather than allowing untrusted users to access source data directly, they are instead provided with a restricted* view*, from which all confidential information has been removed. The program that generates the view effectively embodies a confidentiality policy for the underlying source data. However, this approach has a significant drawback: it prevents users from* updating *the data in the view.*

To address the "view update problem" in general, a number of bidirectional languages *have been proposed. Programs in these languages—often called* lenses*—can be run in two directions: read from left to right, they map sources to views; from right to left, they map updated views back to updated sources. However, existing bidirectional languages do not deal adequately with security. In particular, they do not provide a way to ensure the integrity of source data as it is manipulated by untrusted users of the view.*

We propose a novel framework of secure lenses *that addresses these shortcomings. We enrich the types of basic lenses with equivalence relations capturing notions of confidentiality and integrity, and formulate the essential security conditions as non-interference properties. We then instantiate this framework in the domain of string transformations, developing syntax for* bidirectional string combinators *with* security-annotated regular expressions *as their types.*

1. Introduction

Security views are a widely used mechanism for controlling access to confidential information in databases and other systems that manage structured information. By forcing users to access data via views that only expose public information, data administrators ensure that secrets will not be leaked, even if the users mishandle the data or are malicious. Security views are robust, making it impossible for users to leak the source data hidden by the view, and they are flexible: since they are implemented as arbitrary programs, they can be used to enforce extremely fine-grained access control policies. However, they are not usually updatable—and for good reason! Propagating updates to views made by untrusted users can, in general, alter the source data, including the parts that are hidden by the view.

Still, there are many applications in which having a mechanism for reliably updating security views would be extremely useful. For example, consider Intellipedia, a collaborative data sharing system based on Wikipedia that is used by members of the intelligence community. The data stored in Intellipedia is classified at the granularity of whole documents, but many documents actually contain a mixture of highly classified and less-classified data. In order to give users with low clearances access to the portions of documents they have sufficient clearance to see, documents often have to be regraded: i.e., the highly classified parts need to be erased or redacted, leaving behind a residual document—a security view!—that can be reclassified at a lower level of clearance. Of course (since we are talking about a wiki), we would like the users of these views to be able to make updates—e.g., to correct errors or add new information—and have their changes be propagated back to the original document.

In general, for a view to be updatable, the program that generates it needs to be *bidirectional*. That is, it must not only be able to transform sources to views but also to map updated views back to updated sources. In previous work, we and many others have proposed a family of languages for describing bidirectional transformations, often called *lenses* [19], [8], [7], [21], [37], [41], [26], [9], [24], [35], [17], [23], [30], [28]. Formally, a lens l mapping between a set S of "source" structures and a set V of "views" comprises three functions

$$\begin{array}{rcl} l.\mathit{get} & \in & S \longrightarrow V \\ l.\mathit{put} & \in & V \longrightarrow S \longrightarrow V \\ l.\mathit{create} & \in & V \longrightarrow S \end{array}$$

that obey "round-tripping" laws for every $s \in S$ and $v \in V$.

$$l.\mathit{get}\ (l.\mathit{put}\ v\ s) = v \qquad (\textsc{PutGet})$$

$$l.\mathit{get}\ (l.\mathit{create}\ v) = v \qquad (\textsc{CreateGet})$$

$$l.\mathit{put}\ (l.\mathit{get}\ s)\ s = s \qquad (\textsc{GetPut})$$

The *get* function defines the view and is a total function from S to V. There are two functions that handle updates: the *put* function takes an updated V and the original S and weaves them together to yield a correspondingly modified S, while the *create* function handles the special case where we need to compute an S from a V but have no S to use as the original (it fills in any source data that is not reflected in the view with default values).

1063-6900/09 $25.00 © 2009 IEEE
DOI 10.1109/CSF.2009.25

The round-tripping laws capture fundamental expectations about how the components of a lens should behave; they are closely related to the classical conditions on correct *view update translators* that have been studied in the database literature [1], [12], [22]. The first two laws require that updates to the view must be propagated "exactly"—i.e., given an updated view, the *put* and *create* functions must produce a source that the *get* function maps to the very same view. The third law imposes an additional "stability" constraint, requiring that the *put* function must not change the source at all if the view has not been modified.

Languages for describing these *basic lenses* have been extensively studied in recent years, but none of the languages that have been proposed so far deal adequately with security issues. The critical issue that they fail to address is that many of the natural ways of propagating view updates back to sources alter the source data in ways that violate expectations about its integrity. For example, in the Intellipedia application, the natural way to propagate the deletion of a section of a regraded document is to delete the corresponding section of the original document. But while doing so faithfully reflects the edit made to the view—formally, it satisfies the PUTGET law—it is not necessarily what we want: if the section in the original document contains additional classified data in nested subsections, then deleting the section is almost surely unacceptable—users should not be able to delete data they do not even have sufficient clearance to see!

One can add an additional behavioral law stipulating that propagating updates to the view must not lose *any* of the hidden data in the source. Indeed, this idea has been explored extensively in the context of databases in the so-called *constant complement* approach [1]. The idea is that the source S should be isomorphic to $(V \times C)$, a product consisting of the view V and a "complement" C that contains all of the information in the source not reflected in the view. The *get* function uses the function witnessing the isomorphism to transform the source into a pair, and projects out the first component. The *put* function pairs up the new view with the original complement and applies the witness in the other direction to obtain the updated source. Note that because the *put* function is injective, it necessarily propagates *all* of the information contained in the complement back to the source—i.e., the information in the complement is kept constant.

We can require that lenses hold a complement constant by imposing an additional law requiring that the effect of two *put*s in a row must be the same as just the second one:

$$l.\mathit{put}\ v'\ (l.\mathit{put}\ v\ s) = l.\mathit{put}\ v'\ s \qquad (\text{PUTPUT})$$

(For details about the relationship between this law and constant complement see [19].) Lenses that satisfy PUTPUT are called *very well behaved*. However, requiring that every lens be very well behaved is a draconian restriction—e.g., it rules out conditionals and iteration operators that are indispensable in practice.

So, because we need to allow untrusted users to modify hidden source data through the view, under certain circumstances, we need a simple, declarative way to specify which parts of the source can be affected by view updates and which parts that cannot. Developing a framework in which it is possible to formulate integrity policies like "these sections in the source can be deleted" or "these sections in the view must not be altered (because doing so would have an unacceptable effect on the source)," and verify that lenses obey them, is the goal of this paper.

To this end, we identify a new semantic space of *secure lenses*, in which types not only describe the sets of structures manipulated by the components of lenses, but also capture the notion that certain parts of the source and view represent endorsed data while other parts may be tainted. Semantically, we model these types as sets of structures together with equivalence relations identifying structures that agree on endorsed data. Syntactically, we describe them using *annotated regular types*—regular expressions decorated with annotations drawn from a set of labels representing static levels of integrity. We formulate a condition ensuring the integrity of source data by stipulating a *non-interference* property for the *put* function as an additional behavioral law. This law ensures that if the update to the view does not change high-integrity data in the view then the *put* function does not modify high-integrity data in the source.

Having identified this semantic space of secure lenses, we then demonstrate its applicability by developing a security-enhanced variant of *Boomerang*, a bidirectional language whose primitives are based on finite-state string transductions [7]. We choose to work in the domain of string transformations both because it is interesting in its own right—the Intellipedia application is just one example out of many where one might want an updatable security view of string data—and because it is a fairly simple setting for exploring the pragmatics of secure lenses, while still offering enough structure to raise a host of issues that also come up in richer settings. In particular, the regular-expression-based type system is powerful enough to encode non-recursive XML schemas and is closely related to the full-blown schema languages of XML transformation languages such as XQuery [6], which are based on regular tree automata.

We present refined typing rules for Boomerang's core primitives—atomic lenses for copying, deleting, and filtering strings, and combinators for concatenation, union, iteration, and sequential composition. These typing rules use an information-flow analysis to track dependencies between data in the source and view and guarantee the lens laws, including the new non-interference property for the *put* function. There are some interesting details compared to information-flow type systems for general-purpose languages, since our types describe data schemas at a high level of precision.

So far, we have been talking only about ensuring the integrity of source data. But confidentiality is also interesting in this context: typically the whole reason for defining a security view is to hide certain parts of the source. To the best of our knowledge, none of the previous work on security views has provided a way to formally verify that the information hidden by the view adheres to a declarative confidentiality policy—the query *is* the policy. But, having developed the technical machinery for tracking integrity, it is easy to extend it to track confidentiality as well, and we do so in our information-flow type system for Boomerang. Thus, the actual type system tracks flows of information in two directions, ensuring confidentiality in the forward direction and integrity in the reverse direction.

Tracking information flow using a static type system yields an analysis that is effective but conservative. For example, if the *put* component of a lens ever produces a tainted result, then the type system must classify the source as tainted to ensure the secure lens properties. However, very often there are many inputs that the *put* function can propagate without tainting the source. In the final technical section of the paper, we extend secure lenses with dynamic checks that allow us to test for and detect these situations. These lenses use a combination of static types and dynamic tests to establish the same essential security properties and, in many cases, can be assigned more flexible types.

Our contributions can be summarized as follows:

1. We propose a semantic space of *secure lenses* that extends our previous work on lenses with a type system ensuring the confidentiality and integrity of data in the source. This provides a framework for building reliable and updatable security views.
2. We develop the syntax and semantics of *annotated regular expressions*, which describe sets of strings as well as equivalence relations that encode confidentiality and integrity policies.
3. We instantiate the semantic space of secure lenses with specific *string lens combinators* drawn from the Boomerang language, and we define a bidirectional information-flow type system for these combinators based on annotated regular types.
4. We present an extension to secure lenses that ensures the integrity of source data but replaces some of the static constraints on lens types with dynamic tests.

To save space, proofs are deferred to the long version of this paper, which is available as a technical report [20].

2. Example

Let's warm up with a very small example—much simpler than the Intellipedia application discussed in the introduction, but still rich enough to raise the same essential issues.[1]

1. Interested readers can find prototype code for computing security views of MediaWiki documents in the Boomerang source distribution.

Suppose that the source is an electronic calendar in which certain appointments, indicated by "*", are intended to be private.

```
*08:30 Coffee with Sara (Starbucks)
 10:00 Meeting with Brett (My office)
 12:00 PLClub Seminar (Seminar room)
*15:00 Work out (Gym)
```

Next, suppose that we want to compute a security view where some of the private data is hidden—e.g., perhaps we want to redact the descriptions of the private appointments by rewriting them to BUSY and, at the same time, we also want to erase the location of every appointment.

```
08:30 BUSY
10:00 Meeting with Brett
12:00 PLClub Seminar
15:00 BUSY
```

Or, perhaps, we want to go a step further and erase private appointments completely.

```
10:00 Meeting with Brett
12:00 PLClub Seminar
```

In either case, having computed a security view, we might like to allow colleagues make changes to the public version of our calendar to correct errors and make amendments. For example, here the user of the view has corrected a misspelling by replacing "Brett" with "Brent" and added a meeting with Michael at four o'clock.

```
08:30 BUSY
10:00 Meeting with Brent
12:00 PLClub
15:00 BUSY
16:00 Meeting with Michael
```

The *put* function of the redacting lens combines this new view with the original source and produces an updated source that reflects both changes:

```
*08:30 Coffee with Sara (Starbucks)
 10:00 Meeting with Brent (My office)
 12:00 PLClub (Seminar room)
*15:00 Work out (Gym)
 16:00 Meeting with Michael
```

Although this particular update was handled in a reasonable way, in general, propagating view updates can violate expectations about the handling of hidden data in the source. For example, if the user of the view deletes some appointments,

```
08:30 BUSY
10:00 Meeting with Brent
```

then the source will also be truncated (as it must, to satisfy the PUTGET law):

```
*08:30 Coffee with Sara (Starbucks)
 10:00 Meeting with Brent (My office)
```

From a certain perspective, this is correct—the updated view was obtained by deleting appointments, and the new source is obtained by deleting the corresponding appointments. But if the owner of the source expects the lens to both hide the private data and maintain the integrity of the hidden data, then it is unacceptable for the user of the view to cause some of the hidden data—the description and location of the three o'clock appointment and the location of the noon appointment—to be discarded.

A similar problem arises when the user of the view replaces a private entry with a public one. Consider a private appointment in the source

```
*15:00 Work out (Gym)
```

which maps via *get* to a view:

```
15:00 BUSY
```

If user of the view replaces it with a public appointment (here, they have insisted an important event has precedence)

```
15:00 Distinguished Lecture
```

then the description (`Work out`) and location (`Gym`) associated with the entry in the original source are both lost.

```
 15:00 Distinguished Lecture
```

As these examples demonstrate, to manage security views using lenses reliably, we need mechanisms for tracking the integrity of source data.

Let us consider an attractive—but impossible—collection of guarantees we might like to have. Ideally, the *get* function of the lens would hide the the descriptions of private appointments as well as the location of every appointment, and the *put* function would take *any* updated view and produce an updated source where all of this hidden data is preserved. Sadly, this is not possible: we either need to allow the possibility that certain updates will cause hidden data to be lost, or, if we insist that it must not, then we need to prevent the user of the view from making those updates—e.g., deleting entries and replacing private entries with public ones—in the first place.

Both alternatives can be expressed using the secure lens framework developed in this paper. To illustrate these choices precisely, we need a few definitions. The source and view types of the redacting and erasing lenses are formed out of regular expressions that describe timestamps, descriptions, and locations (along with a few predefined regular expressions, NUMBER, COLON, SPACE, etc.) defined in Boomerang as follows:

```
let TIME : regexp =
  NUMBER{2} . COLON . NUMBER{2} . SPACE
let DESC : regexp =
  [^\n()]* - (ANY . BUSY . ANY)
let LOCATION : regexp =
  (SPACE . LPAREN . [^()]* . RPAREN)?
```

Boomerang uses POSIX notation for character sets (`[^\n()]`) and repetition (`*` and `{2}`). The (`.`) and (`-`) operators denote concatenation and difference.

To specify the policy that prevents the user from applying updates to the view that would cause hidden data to be lost, we pick a type that marks some of the data as endorsed by decorating the bare regular expressions with annotations. Here is a type in which the private appointments are endorsed, as indicated by annotations of the form $(R : \mathsf{E})$, but the public appointments are tainted, as indicated by annotations of the form $(R{:}\mathsf{T})$:

$$
\begin{array}{l}
(\ (\mathtt{SPACE}\cdot\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{LOCATION}\cdot\mathtt{NEWLINE}){:}\,\mathsf{T}\\
\mid (\mathtt{ASTERISK}\cdot\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{LOCATION}\cdot\mathtt{NEWLINE}){:}\,\mathsf{E})^{*}\\
\Longleftrightarrow\\
((\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{NEWLINE}){:}\,\mathsf{T} \mid (\mathtt{TIME}\cdot\mathtt{BUSY}\cdot\mathtt{NEWLINE}){:}\,\mathsf{E})^{*}
\end{array}
$$

As described in the next section, before the owner of the source data allows the user of the view to propagate their updates back to the source using the *put* function, they check that the original and updated views agree on endorsed data. In this case, since the private appointments are endorsed, they will refuse to propagate views where the private appointments have been modified. (The public appointments, however, may be freely modified.)

Alternatively, to specify the policy that provides weaker guarantees about the integrity of source data but allows more updates, we pick a type that labels both public and private appointments as tainted:

$$
\begin{array}{l}
(\ (\mathtt{SPACE}\cdot\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{LOCATION}\cdot\mathtt{NEWLINE}){:}\,\mathsf{T}\\
\mid (\mathtt{ASTERISK}\cdot\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{LOCATION}\cdot\mathtt{NEWLINE}){:}\,\mathsf{T})*\\
\Longleftrightarrow\\
(((\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{NEWLINE}) \mid (\mathtt{TIME}\cdot\mathtt{BUSY}\cdot\mathtt{NEWLINE})){:}\,\mathsf{T})*
\end{array}
$$

With this type, the user of the view may update the view however they like—the whole view is tainted—but the lens does not guarantee the integrity of any appointments in the source. The fact that the entire source may be tainted is reflected explicitly in its type.

Here is the Boomerang code that implements these lenses.

```
let public : lens =
  del SPACE .
  copy ( TIME . DESC ) .
  del LOCATION .
  copy NEWLINE

let private : lens =
  del ASTERISK .
  copy TIME .
  ( ( DESC . LOCATION ) <-> "BUSY" ) .
  copy NEWLINE
```

```
let redact : lens =
  public* . ( private . public* )*

let erase : lens =
  filter (stype public) (stype private);
  public*
```

Note that there are no security type annotations in these programs—the current Boomerang implementation only tracks basic lens types, leaving security type annotations to be checked by hand. We plan to extend the implementation with annotated types in the near future.

In the forward direction, these lens definitions can be read as ordinary string transducers, written in regular expression style. For example, the `public` lens deletes a space character (`del SPACE`), copies a timestamp and description (`copy (TIME . DESC)`), deletes an optional location (`del LOCATION`), and copies a newline character (`copy NEWLINE`). The concatenation operator (`.`) combines lenses in the obvious way. Similarly, the `private` lens deletes an asterisk (`del ASTERISK`), copies the timestamp (`copy TIME`), redacts the event description and optional location by rewriting them to `BUSY` (`DESC . LOCATION? <-> "BUSY"`), and copies a newline (`copy NEWLINE`). The top-level `redact` lens processes blocks of public appointments (`public*`) interspersed with private appointments (`private`). The iteration operator (`*`) works by splitting the source string into a list of substrings and processing these substrings using the *get* component of the lens being iterated. The top-level `erase` lens processes the source in two phases, combined sequentially (`;`). It filters away the private entries in the first phase (`stype` extracts the source type of a lens), and transforms the public appointments (`public*`) in the second phase.

In the reverse direction, these definitions can be read as functions that combine the new view with the original source, propagating the information contained in the view and restoring discarded information from the source. For example, the *put* function of the `copy` lens (which does not discard information in the *get* direction) simply copies the new view to the source. On the other hand, the *put* functions of the delete (`del`) and rewriting (`<->`) lenses, which discard the entire source in the *get* direction, restore the original source. The concatenation (`.`) and iteration (`*`) operators split the source and view in substrings and apply the *put* of their sublenses to pairs of these smaller strings.[2]

2. The *put* functions of the lenses we consider here operate *positionally*—e.g., the *put* function of $l*$ splits the source and view into substrings and applies $l.put$ to pairs of these in order. It is not hard to think of examples where this behavior is not what is wanted (imagine deleting the first element of the view...). Readers familiar with Boomerang may recall that it is actually based on *dictionary lenses*, which incorporate mechanisms for handling ordered data [7]. It would be interesting to develop a secure version of dictionary lenses, but we leave this extension for future work.

3. Semantics

The behavioral laws obeyed by basic lenses ensure some fundamental sanity conditions, but, as we saw in the preceding section, to uses lenses reliably in security applications we need additional guarantees. In this section, we present a refined semantics for *secure lenses* that obey new behavioral laws stipulating that the *put* function must not taint endorsed (high integrity) source data and the *get* function must not leak secret (high confidentiality) data.

Let $\mathcal{P}$ (for "privacy") and $\mathcal{Q}$ (for "quality") be lattices of security labels representing levels of confidentiality and integrity, respectively. To streamline the presentation, we will mostly work with two-point lattices $\mathcal{P} = \{\mathsf{P}, \mathsf{S}\}$ (for "public" and "secret") with $\mathsf{P} \sqsubseteq \mathsf{S}$ and $\mathcal{Q} = \{\mathsf{E}, \mathsf{T}\}$ (for "endorsed" and "tainted") with $\mathsf{E} \sqsubseteq \mathsf{T}$.

$$\mathcal{P} = \begin{tikzcd} \mathsf{S} \arrow[d, dash] \\ \mathsf{P} \end{tikzcd} \qquad \mathcal{Q} = \begin{tikzcd} \mathsf{T} \arrow[d, dash] \\ \mathsf{E} \end{tikzcd}$$

(Although we call endorsed data "high integrity" informally, it is actually the least element in $\mathcal{Q}$. This is standard—intuitively, data that is higher in the lattice needs to be handled more carefully while data that is lower in the lattice can be used more flexibly.) Our results generalize straightforwardly to arbitrary finite lattices.

Fix sets S (of sources) and V (of views). To formalize notions like "these two sources contain the same public information (but possibly differ on their private parts)," we will use equivalence relations on S and V indexed by both lattices of security labels. Formally, let $\sim_k^S \subseteq S \times S$ and $\sim_k^V \subseteq V \times V$ be families of equivalence relations indexed by security labels in $\mathcal{P}$, and let $\approx_k^S \subseteq S \times S$ and $\approx_k^V \subseteq V \times V$ be families of equivalence relations indexed by labels in $\mathcal{Q}$. In what follows, when S and V are clear from context, we will suppress the superscripts to lighten the notation. Typically, $\sim_{\mathsf{S}}$ and $\approx_{\mathsf{T}}$ will be equality, while $\sim_{\mathsf{P}}$ and $\approx_{\mathsf{E}}$ will be coarser relations that identify sources and views containing the same public and endorsed parts, respectively. These equivalences capture confidentiality and integrity policies for the data.

A *secure lens* l has three components

$$\begin{aligned} l.get &\in S \longrightarrow V \\ l.put &\in V \longrightarrow S \longrightarrow S \\ l.create &\in V \longrightarrow S \end{aligned}$$

that obey the following laws for every s in S, v in V, and k in $\mathcal{Q}$ or $\mathcal{P}$ as appropriate:

$$l.get\ (l.put\ v\ s) = v \qquad (\text{PutGet})$$

$$l.get\ (l.create\ v) = v \qquad (\text{CreateGet})$$

$$\frac{v \approx_k l.get\ s}{l.put\ v\ s \approx_k s} \qquad (\text{GetPut})$$

$$\frac{s \sim_k s'}{l.get\ s \sim_k l.get\ s'} \qquad (\text{GetNoLeak})$$

The PUTGET and CREATEGET laws here are identical to the basic lens version that we saw in the Introduction and express the same fundamental constraint on lenses: updates to views must be propagated to sources exactly.

The GETPUT law for secure lenses, however, is different. It ensures the integrity of source data, expressed as a non-interference condition on the *put* function. Formally, it requires that if the original view (i.e., the one computed from the original source) and the new view are related by $\approx_k$, then the original source and the updated source computed by *put* must also be related by $\approx_k$. For example, if the original and new view are related by $\approx_{\mathsf{E}}$—i.e., they agree on the endorsed data—then GETPUT guarantees that the new source will also agree with the original source on endorsed data. Note that we recover the basic lens law GETPUT when $\approx_k$ is equality, as it typically is for $\approx_{\mathsf{T}}$.

The GETPUT law suggests a protocol for using secure lenses: before the owner of the source allows the user of a view to invoke the *put* function, they check that the original and updated views are related by $\approx_k$ for every k that is lower in $\mathcal{Q}$ than the data the user is allowed to edit—e.g., in the two-point lattice, a user whose edits are considered tainted would have the checks performed using $\approx_{\mathsf{E}}$. The owner of the source only performs the *put* if the test succeeds.

Secure lenses obey a variant of the PUTPUT law capturing a notion of lenses that are very well behaved on endorsed data. The following PUTPUTENDORSED law can be derived from GETPUT and PUTGET:

$$\frac{v' \approx_k l.get\ s \approx_k v}{l.put\ v'\ (l.put\ v\ s) \approx_k l.put\ v'\ s} \quad \text{(PUTPUTENDORSED)}$$

If $\approx_k$ is equality (as it typically is for $\approx_{\mathsf{T}}$) then PUTPUTENDORSED reduces to GETPUT law: it says that applying *put* (twice) to the view obtained by invoking *get* on the source yields the original source (both times). If, however, $\approx_k$ relates strings that agree on endorsed data (as it typically does for $\approx_{\mathsf{E}}$) then PUTPUTENDORSED implies that *put* must preserve the endorsed hidden data in the source. This law allows operators such as conditional and iteration whose *put* functions do sometimes discard hidden source data in the reverse direction, and are therefore not very well behaved lenses in the strict sense, as long as they indicate that they do so in their type, by marking the source data that may be discarded as tainted.

Our main concern in this paper is preserving integrity after updates, but it is also worth noticing that we can tell an improved story about confidentiality. In previous work on (non-updatable) security views, the confidentiality policy enforced by the view is not stated explicitly—the private information in the source is simply "whatever information is projected away in the view." Our security lenses, on the other hand, have an explicit representation of confidentiality policies, embodied in the choice of equivalence relations. Thus, we can add the GETNOLEAK law stipulating that the *get* function must not leak confidential source information source. This law is formulated as a non-interference condition stating that, if two sources are related by $\sim_k$, then the results computed by *get* must also be related by $\sim_k$. For example, when $\sim_{\mathsf{P}}$ relates two sources, GETNOLEAK ensures that the views computed from those sources also agree on public data. Thus, secure lenses provide a confidentiality guarantee that can be understood without having to look at the program defining the lens.[3] In the next section, we present a declarative language for security annotations that can be used to describe many such equivalences.

4. Annotated Regular Expressions

The types of our secure string lens combinators are regular expressions annotated with labels drawn from the two lattices of security labels. In this section, we define the precise syntax and semantics of these annotated regular expressions.

We begin by fixing a few pieces of notation. Let Σ be a finite alphabet (e.g., ASCII). A language L is a subset of Σ^*. When L is non-empty, we write $rep(L)$ for an arbitrary representative of L. The ϵ symbol denotes the empty string, and $u{\cdot}v$ denotes the concatenation of strings u and v. We lift concatenation to languages in the obvious way. The iteration of L is $L^* = \bigcup_{n=0}^{\infty} L^n$, where L^n denotes the n-fold concatenation of L with itself.

Many of our definitions require that every string in the concatenation of two languages have a unique factorization. We say that two languages L_1 and L_2 are unambiguously concatenable, written $L_1 {\cdot^!} L_2$, if for every u_1, v_1 in L_1 and u_2, v_2 in L_2, if $(u_1{\cdot}u_2) = (v_1{\cdot}v_2)$ then $u_1 = v_1$ and $u_2 = v_2$. Similarly, we say that a language L is unambiguously iterable, written $L^{!*}$, if for every $(u_1, \ldots, u_m), (v_1, \ldots, v_n), \in L$, if $(u_1 \cdots u_m) = (v_1 \cdots v_n)$ then $m = n$ and $u_i = v_i$ for $i \in \{1...n\}$. It is decidable whether two regular languages are unambiguously concatenable and whether a language is unambiguously iterable (see [4, Proposition 4.1.3]).

Now we are ready to define our types. Let $\mathcal{K} = (K, \sqsubseteq)$ be a finite lattice.[4] The set of *annotated regular expressions* over Σ and $\mathcal{K}$ is given by the following grammar

$$\mathcal{R} ::= \emptyset \mid u \mid \mathcal{R}{\cdot}\mathcal{R} \mid \mathcal{R}|\mathcal{R} \mid \mathcal{R}^* \mid \mathcal{R}{:}k$$

where $u \in \Sigma^*$ and $k \in K$. An annotated expression can be interpreted in two ways:

3. We treat confidentiality and integrity as orthogonal—almost, see Section 6—so users can also choose $\sim_{\mathsf{P}}^{S}$ to be equality and our laws place no constraints on confidentiality. This yields the same story as in previous systems, where "what the view hides" is read off from the view definition.

4. Note that, to streamline the notation, annotations here are drawn from just *one* lattice of labels. Later, when we use these annotated regular expressions to denote the types of secure string lenses, we'll decorate them with labels from both $\mathcal{P}$ and $\mathcal{Q}$. When we calculate the semantics of a type—in particular, the equivalence relations it denotes—we will consider each lattice separately, ignoring the labels in the other lattice.

- As a regular language $\mathcal{L}(R)$, defined in the usual way (after ignoring annotations).
- As a family of equivalence relations $\sim_k \subseteq (\mathcal{L}(R) \times \mathcal{L}(R))$ capturing the intuitive notion that two strings differing only in high-security regions cannot be distinguished by a low-security observer.

To lighten the notation, when it is clear from context we will often conflate R and $\mathcal{L}(R)$—e.g., we will write $u \in R$ instead of $u \in \mathcal{L}(R)$.

In many languages with security-annotated types, the type structure of the language is relatively simple and so the definition of the "observability relations" is straightforward. However, annotated regular expressions have features like non-disjoint unions that make the intended semantics less obvious—indeed, there seem to be several reasonable alternatives. We describe here a simple semantics based on a notion of erasing inaccessible substrings that we find natural and discuss alternatives toward the end of the section.

Formally, we define the equivalence relations using a function that erases substrings that are inaccessible to a k-observer and take a pair of strings to be equivalent if their erased versions are identical. For ease of exposition, we will describe the erasing function as the composition of two functions: one that marks the inaccessible regions of a string and another that erases marked regions. Let $\#$ be a fresh symbol, $hash(R)$ be the function that transforms strings in $\mathcal{L}(R)$ by rewriting every character to $\#$,

$$hash(R)(u) \triangleq \underbrace{\#\cdots\#}_{|u| \text{ times}}$$

and let $mark(R, k)$ be a relation that marks inaccessible characters:

$$\begin{aligned}
mark(\emptyset, k) &\triangleq \{\} \\
mark(u, k) &\triangleq \{(u, u)\} \\
mark(R_1 \cdot R_2, k) &\triangleq mark(R_1, k) \cdot mark(R_2, k) \\
mark(R_1 \mid R_2, k) &\triangleq mark(R_1, k) \;\&\; (\mathcal{L}(R_1) \setminus \mathcal{L}(R_2)) \\
&\quad \cup\, mark(R_2, k) \;\&\; (\mathcal{L}(R_2) \setminus \mathcal{L}(R_1)) \\
&\quad \cup\, mark(R_1, k) \;\&\; mark(R_2, k) \\
mark(R_1{}^*, k) &\triangleq mark(R_1, k)^* \\
mark(R_1{:}j, k) &\triangleq \begin{cases} mark(R_1, k) & \text{if } k \sqsupseteq j \\ hash(R_1) & \text{otherwise} \end{cases}
\end{aligned}$$

The definition of $mark$ uses the operations of union, concatenation, and iteration, which we lift to relations in the obvious way. The most interesting case is for union. In general, the languages denoted by a pair of annotated regular expressions can overlap, so we need to specify how to mark strings that are described by both expressions as well as strings that are only described by one of the expressions. There are three cases: To handle the strings described by only one of the expressions, we use an intersection operator that restricts a marking relation Q to a regular language L:

$$Q \;\&\; L \triangleq \{(u, v) \mid (u, v) \in Q \wedge u \in L\}$$

To handle strings described by both expressions, we use an intersection operator that merges markings

$$Q_1 \;\&\; Q_2 \triangleq \{(u, merge(v_1, v_2)) \mid (u, v_i) \in Q_i\},$$

where:

$$\begin{aligned}
merge(\epsilon, \epsilon) &= \epsilon \\
merge(\#\cdot v_1, \_\cdot v_2) &= \#\cdot merge(v_1, v_2) \\
merge(\_\cdot v_1, \#\cdot v_2) &= \#\cdot merge(v_1, v_2) \\
merge(c\cdot v_1, c\cdot v_2) &= c\cdot merge(v_1, v_2).
\end{aligned}$$

The effect is that characters marked by either relation are marked in the result.

Although $mark$ is a relation in general, we are actually interested in cases where it is a function. Unfortunately, the operations of concatenation, and iteration used in the definition of $mark$ do not yield a function in general due to ambiguity. Thus, we impose the following condition:

4.1 Definition: R is *well-formed* iff every subexpression of the form $R_1 \cdot R_2$ is unambiguously concatenable ($\mathcal{L}(R_1) \cdot^! \mathcal{L}(R_2)$) and every subexpression of the form R^* is unambiguously iterable ($\mathcal{L}(R)^{!*}$).

4.2 Proposition: If R is well formed, then $mark(R, k)$ is a function.

In what follows, we will tacitly assume that all annotated expressions under discussion are well formed. (And when we define typing rules for our secure lens combinators, below, we will be careful to ensure well-formedness.)

Let $erase$ be the function on $(\Sigma \cup \{\#\})$ that copies characters in Σ and erases $\#$ symbols. We define $\sim_k$ as the relation induced by marking and then erasing:

$$\begin{aligned}
hide_k(u) &\triangleq erase(mark(R, k)(u)) \\
\sim_k &\triangleq \{(u, v) \mid hide_k(u) = hide_k(v)\}
\end{aligned}$$

It is easy to see that $\sim_k$ is an equivalence relation.

4.3 Lemma: Let R_1 and R_2 be well-formed annotated regular expressions over a finite lattice $\mathcal{K}$. It is decidable whether R_1 and R_2 are equivalent.

Proof sketch: Equivalence for the regular languages $\mathcal{L}(R_1)$ and $\mathcal{L}(R_2)$ is straightforward. Moreover, each relation $\sim_k$ is induced by $hide_k(-)$, which is definable as a rational function—a class for which equivalence is decidable [3, Chapter IV, Corollary 1.3]. $\square$

As examples to illustrate the semantics, consider a two-point lattice $(\{\mathsf{P}, \mathsf{S}\}, \sqsubseteq)$ with $\mathsf{P} \sqsubseteq \mathsf{S}$ and take R_1 to be the annotated expression `[a-z]`$:\mathsf{S}$. Then for every string u in $\mathcal{L}(R_1)$ we have $mark(R_1, \mathsf{P})(u) = \#$, and so $hide_{\mathsf{P}}(u) = \epsilon$, and $\sim_{\mathsf{P}}$ is the total relation. For the annotated relation $R_1{}^*$, the equivalence $\sim_{\mathsf{P}}$ is again the total relation because every u in $\mathcal{L}(R_1{}^*)$ maps to a sequence of $\#$ symbols by

$mark(R_1{}^*, \mathsf{P})$, and so $hide_{\mathsf{P}}(u) = \epsilon$. More interestingly, for R_2 defined as

$$(\texttt{[a-z]:P})\cdot(\texttt{[0-4]:S}) \mid (\texttt{[a-z]:P})\cdot(\texttt{[5-9]:S}),$$

and any string $c{\cdot}n$ in $\mathcal{L}(R_2)$ we have $mark(R_2, \mathsf{P})(c{\cdot}n) = c\#$ and so $hide_{\mathsf{P}}(c{\cdot}n) = c$. It follows that $cn \sim_{\mathsf{P}} c'n'$ iff $c = c'$. Finally, for $R_2{}^*$ the equivalence $\sim_{\mathsf{P}}$ identifies $(c_1{\cdot}n_1 \cdots c_i{\cdot}n_i)$ and $(d_1{\cdot}m_1 \cdots d_j{\cdot}m_j)$ iff $i = j$ and $c_i = d_i$ for i from 1 to n.

As we remarked above, there are other reasonable ways to define $\sim_k$. For example, instead of marking and erasing, we could instead compose $mark$ with a function that compresses sequences of # symbols into a single #. The equivalence induced by this function would allow low-security observers to determine the presence and location of high-security data, but would obscure its content. We could even take the equivalence induced by the $mark$ function itself! This semantics would reveal the presence, location, and length of high-security data to low-security observers. There may well be scenarios where one of these alternative semantics more accurately models the capabilities of low-security observers. For simplicity, we will use the erasing semantics in the rest of the paper.

5. Secure String Lens Combinators

We now turn to defining secure versions of the core lenses found in Boomerang [7]. The functional components of these secure lenses are identical to their basic lens versions, but their typing rules are enhanced with an information-flow analysis that guarantees the secure lens laws.

Copy The simplest lens, *copy E*, takes a well-formed annotated regular expression as an argument. It copies strings belonging to E in both directions.

$$\boxed{\begin{array}{c}\dfrac{E \text{ well-formed}}{\textit{copy } E \in E \Longleftrightarrow E}\\[2ex] \begin{array}{ll}\textit{get } s & = s\\ \textit{put } v\ s & = v\\ \textit{create } v & = v\end{array}\end{array}}$$

The rule of inference at the top of this box should be read as a lemma asserting that if E is well-formed then $(\textit{copy } E)$ is a well-behaved secure lens at $E \Longleftrightarrow E$: i.e., its components are total functions that obey the PUTGET, CREATEGET, GETPUT, and GETNOLEAK laws.

Const The next lens, *const*, takes as arguments two well-formed annotated regular expressions E and F, with F a singleton, and a string d that belongs to E. It maps every source string in E to the unique element of F in the *get* direction, and restores the discarded source string in the reverse direction. The d argument is used as the default source by the *create* function.

$$\boxed{\begin{array}{c}\dfrac{E, F \text{ well-formed} \qquad |F| = 1 \qquad d \in E}{\textit{const } E\ F\ d \in E \Longleftrightarrow F}\\[2ex] \begin{array}{ll}\textit{get } s & = \textit{rep}(F)\\ \textit{put } v\ s & = s\\ \textit{create } v & = d\end{array}\end{array}}$$

Typically F will just be a bare string u, but occasionally it will be useful to decorate it with integrity labels (e.g., see the discussion following the union combinator below). The typing rule for *const* places no additional labels on the source and view types. This is safe: the *get* function maps every string in E to $\textit{rep}(F)$, so GETNOLEAK holds trivially. The *put* restores the source exactly—including any high-integrity data—so GETPUT also holds trivially. Using *const*, we can define some additional lenses as derived forms:

$$\begin{array}{rcll} E \leftrightarrow F & \triangleq & \textit{const } E\ F\ \textit{rep}(E) & \in E \Longleftrightarrow F\\ \textit{del } E & \triangleq & E \leftrightarrow \epsilon & \in E \Longleftrightarrow \epsilon\\ \textit{ins } F & \triangleq & \epsilon \leftrightarrow F & \in \epsilon \Longleftrightarrow F \end{array}$$

$E \leftrightarrow F$ is like *const* but chooses an arbitrary element of E as the default; *del E* deletes a source string belonging to E in the *get* direction and restores it in the *put* direction; *ins F* inserts the fixed string $\textit{rep}(F)$ in the *get* direction and removes it in the *put* direction.

Union The union combinator behaves like a conditional operator on lenses. Its typing rule uses some new notation, which will be explained shortly.

$$\boxed{\begin{array}{c}\dfrac{\begin{array}{c}(S_1 \cap S_2) = \emptyset\\ l_1 \in S_1 \Longleftrightarrow V_1 \qquad l_2 \in S_2 \Longleftrightarrow V_2\\ q = \bigvee\{k \mid k \text{ min obs. } V_1 \neq V_2 \wedge V_1 \ \&\ V_2 \text{ agree}\}\\ p = \bigvee\{k \mid k \text{ min obs. } (S_1 \cap S_2) = \emptyset\}\end{array}}{l_1 \mid l_2 \in (S_1 \mid S_2){:}q \Longleftrightarrow (V_1 \mid V_2){:}p}\\[3ex] \textit{get } s = \begin{cases} l_1.\textit{get } s & \text{if } s \in S_1\\ l_2.\textit{get } s & \text{if } s \in S_2\end{cases}\\[2ex] \textit{put } v\ s = \begin{cases} l_1.\textit{put } v\ s & \text{if } s \in S_1 \wedge v \in V_1\\ l_2.\textit{put } v\ s & \text{if } s \in S_2 \wedge v \in V_2\\ l_1.\textit{create } v & \text{if } s \in S_2 \wedge v \in (V_1 \setminus V_2)\\ l_2.\textit{create } v & \text{if } s \in S_1 \wedge v \in (V_2 \setminus V_1)\end{cases}\\[2ex] \textit{create } v = \begin{cases} l_1.\textit{create } v & \text{if } v \in V_1\\ l_2.\textit{create } v & \text{if } v \in (V_2 \setminus V_1)\end{cases}\end{array}}$$

In the forward direction, the union lens uses a membership test on the source string to select a lens. As usual with conditionals, the typing rule for union needs to be designed carefully to take implicit flows of confidential information into account. To see why, consider the union of the following two lenses:

$$\begin{array}{rcll} l_1 & \triangleq & \texttt{[0-4]:S} \leftrightarrow \texttt{A} & \in (\texttt{[0-4]:S}) \Longleftrightarrow \texttt{A}\\ l_2 & \triangleq & \texttt{[5-9]:S} \leftrightarrow \texttt{B} & \in (\texttt{[5-9]:S}) \Longleftrightarrow \texttt{B} \end{array}$$

We might be tempted to assign it the type obtained by taking the unions of the source and view types of the smaller lenses:

$$(l_1 \mid l_2) \in (\texttt{[0-4]}{:}\mathsf{S} \mid \texttt{[5-9]}{:}\mathsf{S}) \Longleftrightarrow (\texttt{A} \mid \texttt{B})$$

But this would be wrong: in general, the *get* function leaks information about which branch was selected, as demonstrated by the following counterexample to GETNOLEAK. By the semantics of annotated regular expressions, we have $\texttt{0} \sim_{\mathsf{P}} \texttt{5}$, since $hide_{\mathsf{P}}$ maps both to the empty string. But:

$$(l_1 \mid l_2).get\ \texttt{0} = \texttt{A} \not\sim_{\mathsf{P}} \texttt{B} = (l_1 \mid l_2).get\ \texttt{5}$$

Most languages with information-flow type systems deal with these implicit flows by raising the security level of the result. Formally, they escalate the label on the type of the result by joining it with the label of the data used in the conditional test. Our typing rule for the union lens is based on this idea, although the computation of the label is somewhat complicated because the conditional test is membership in S_1 or S_2, so "the label of the data used in the conditional test" is the least label that can distinguish strings in S_1 from those in S_2. Returning to our example with $(l_1 \mid l_2)$ and the two-point lattice, S is the only such label, so we label the entire view as secret.

For annotated regular expressions, we can decide whether a given label distinguishes strings in S_1 from those in S_2, and so we can compute the least such label (as $\mathcal{P}$ is finite). Let k be a label in $\mathcal{P}$. We say that k *observes* $(S_1 \cap S_2) = \emptyset$ iff for every string $s_1 \in S_1$ and $s_2 \in S_2$ we have $s_1 \not\sim_k s_2$. Note that k observes $(S_1 \cap S_2) = \emptyset$ iff the codomains of the rational function $hide_k(-)$ for S_1 and S_2 are disjoint. As the codomain of a rational function is computable and a regular language, we can decide whether k observes the disjointness of S_1 and S_2. In a general lattice there may be several labels that observe the disjointness of S_1 and S_2. The label p we compute for the view type is the join of the set of minimal labels that observe their disjointness.

In the *put* direction, the union lens selects a lens using membership tests on the source and the view (the test on the view takes priority, with the test on the source breaking any ties). Here we need to consider the integrity of the source data since modifying the view can result in l_2 being used for the *put* function even though l_1's *get* function was used to generate the original view, or vice versa. To safely handle these situations, we need to treat the source string as more tainted. For example, consider the union of:

$$l_1 \triangleq (del\ \texttt{[0-4]}{:}\mathsf{E}) {\cdot} (copy\ \texttt{[A-Q]}{:}\mathsf{T}) \\ \in (\texttt{[0-4]}{:}\mathsf{E} {\cdot} \texttt{[A-Q]}{:}\mathsf{T}) \Longleftrightarrow (\texttt{[A-Q]}{:}\mathsf{T})$$

$$l_2 \triangleq (del\ \texttt{[5-9]}{:}\mathsf{E}) {\cdot} (copy\ \texttt{[F-Z]}{:}\mathsf{T}) \\ \in (\texttt{[5-9]}{:}\mathsf{E} {\cdot} \texttt{[F-Z]}{:}\mathsf{T}) \Longleftrightarrow (\texttt{[F-Z]}{:}\mathsf{T})$$

This lens does not have secure lens type obtained by taking the union of the source and view types

$$(l_1 \mid l_2) \in (\texttt{[0-4]}{:}\mathsf{E} {\cdot} \texttt{[A-Q]}{:}\mathsf{T}) \mid (\texttt{[5-9]}{:}\mathsf{E} {\cdot} \texttt{[F-Z]}{:}\mathsf{T}) \\ \Longleftrightarrow (\texttt{[A-Q]}{:}\mathsf{T} \mid \texttt{[F-Z]}{:}\mathsf{T})$$

because the *put* function sometimes fails to maintain the integrity of the number in the source, as demonstrated by the following counterexample to GETPUT. By the semantics of annotated regular expressions, we have $\texttt{Z} \approx_{\mathsf{E}} \texttt{A}$, since $hide_{\mathsf{E}}$ maps both to the empty string. But

$$(l_1 \mid l_2).put\ \texttt{Z}\ \texttt{0A} = \texttt{5Z} \not\approx_{\mathsf{E}} \texttt{0A}$$

To obtain a sound typing rule for union, we need to raise the integrity label on the source—i.e., consider the source more tainted. We do this by annotating the source type with the least label q such that we can transform a string belonging to $(V_1 \setminus V_2)$ to a string belonging to V_2 (or vice versa) by modifying q-tainted data.

Formally, we compute q as the join of the minimal set of labels in $\mathcal{Q}$ that observe that V_1 and V_2 are not identical—e.g., for the lens above, T. For technical reasons—to ensure that $v \in V_1$ and $s \in S_1$ and $v \approx_k^{(S_1|S_2)} (l_1 \mid l_2).get\ s$ implies $v \approx_k^{S_1} l_1.get\ s$—we also require that q observe that V_1 and V_2 denote the same equivalence relations on strings in their intersection; we write this condition as "V_1 & V_2 agree." Both of these properties can be decided for annotated regular expressions using elementary constructions.

An important special case arises when V_1 and V_2 coincide. Then, since both lenses are capable of handling the entire view type, the same lens is always selected for *put* as was selected for *get*. For example, the union of

$$l_1 \triangleq (del\ \texttt{[0-4]}{:}\mathsf{E}) {\cdot} (copy\ \texttt{[A-Z]}{:}\mathsf{T}) \\ \in (\texttt{[0-4]}{:}\mathsf{E} {\cdot} \texttt{[A-Z]}{:}\mathsf{T}) \Longleftrightarrow (\texttt{[A-Z]}{:}\mathsf{T})$$

$$l_2 \triangleq (del\ \texttt{[5-9]}{:}\mathsf{E}) {\cdot} (copy\ \texttt{[A-Z]}{:}\mathsf{T}) \\ \in (\texttt{[5-9]}{:}\mathsf{E} {\cdot} \texttt{[A-Z]}{:}\mathsf{T}) \Longleftrightarrow (\texttt{[A-Z]}{:}\mathsf{T})$$

does have the type:

$$(\texttt{[0-4]}{:}\mathsf{E} {\cdot} \texttt{[A-Z]}{:}\mathsf{T}) \mid (\texttt{[5-9]}{:}\mathsf{E} {\cdot} \texttt{[A-Z]}{:}\mathsf{T}) \Longleftrightarrow \texttt{[A-Z]}{:}\mathsf{T}$$

Our typing rule captures this case: if $V_1 = V_2$ then q is the join of the empty set, which equals the minimal element E. Annotating with E, the least element in $\mathcal{Q}$, is semantically equivalent to having no annotation at all.

Concatenation The next operator takes two lenses and forms a lens that operates on the concatenations of their source and view types.

$$\boxed{\begin{array}{c} l_1 \in S_1 \Longleftrightarrow V_1 \quad S_1 {\cdot}^! S_2 \\ l_2 \in S_2 \Longleftrightarrow V_2 \quad V_1 {\cdot}^! V_2 \\ q = \bigvee\{k \mid k \text{ min obs. } V_1 {\cdot}^! V_2\} \\ p = \bigvee\{k \mid k \text{ min obs. } S_1 {\cdot}^! S_2\} \\ \hline l_1 {\cdot} l_2 \;\in\; (S_1 {\cdot} S_2){:}q \Longleftrightarrow (V_1 {\cdot} V_2){:}p \\ \\ \begin{array}{lcl} get\ (s_1 {\cdot} s_2) & = & (l_1.get\ s_1) {\cdot} (l_2.get\ s_2) \\ put\ (v_1 {\cdot} v_2)\ (s_1 {\cdot} s_2) & = & (l_1.put\ v_1\ s_1) {\cdot} (l_2.put\ v_2\ s_2) \\ create\ (v_1 {\cdot} v_2) & = & (l_1.create\ v_1) {\cdot} (l_2.create\ v_2) \end{array} \end{array}}$$

The *get* function takes the source string, splits it in two, applies the *get* component of l_1 and l_2 to these smaller strings, and concatenates the result. We write $(s_1{\cdot}s_2)$ to indicate that s_1 and s_2 are the unique substrings belonging to S_1 and S_2 (which are unambiguously concatenable).

As with the union lens, the typing rule for concatenation also needs to be designed carefully to take implicit flows of information into account. Here the implicit flows stem from the way that the concatenation operator splits strings. As an example, consider a lens l_1 that maps `a0` to `A` and `a1` to `a`, and a lens l_2 that maps `b0` to `B` and `b1` to `b`, where all of the source data is private except for the `1`, which is public:

$$\begin{aligned} l_1 &\triangleq ((\mathtt{a}{:}\mathsf{S}){\cdot}(\mathtt{1}{:}\mathsf{P}) \leftrightarrow \mathtt{A}) \mid ((\mathtt{a}{:}\mathsf{S}){\cdot}(\mathtt{0}{:}\mathsf{S}) \leftrightarrow \mathtt{a}) \\ &\in (\mathtt{a}{:}\mathsf{S}{\cdot}(\mathtt{0}{:}\mathsf{S} \mid \mathtt{1}{:}\mathsf{P})) \Longleftrightarrow (\mathtt{A} \mid \mathtt{a}) \\ l_2 &\triangleq ((\mathtt{b}{:}\mathsf{S}){\cdot}(\mathtt{1}{:}\mathsf{P}) \leftrightarrow \mathtt{B}) \mid ((\mathtt{b}{:}\mathsf{S}){\cdot}(\mathtt{0}{:}\mathsf{S}) \leftrightarrow \mathtt{b}) \\ &\in (\mathtt{b}{:}\mathsf{S}{\cdot}(\mathtt{0}{:}\mathsf{S} \mid \mathtt{1}{:}\mathsf{P})) \Longleftrightarrow (\mathtt{B} \mid \mathtt{b}) \end{aligned}$$

The concatenation of l_1 and l_2 does not have the type obtained by concatenating their source and view types,

$$\begin{aligned} l_1{\cdot}l_2 &\in ((\mathtt{a}{:}\mathsf{S}{\cdot}(\mathtt{0}{:}\mathsf{S} \mid \mathtt{1}{:}\mathsf{P})){\cdot}(\mathtt{b}{:}\mathsf{S}{\cdot}(\mathtt{0}{:}\mathsf{S} \mid \mathtt{1}{:}\mathsf{P}))) \\ &\Longleftrightarrow ((\mathtt{A} \mid \mathtt{a}){\cdot}(\mathtt{B} \mid \mathtt{b})), \end{aligned}$$

because the *get* function exposes the way that the source string was split, as demonstrated by a counterexample to GETNOLEAK:

$$\begin{array}{ll} & \mathtt{a1b0} \sim_{\mathsf{P}} \mathtt{a0b1} \\ \text{but} & (l_1{\cdot}l_2).\mathit{get}\ \mathtt{a1b0} = \mathtt{Ab} \not\sim_{\mathsf{P}} \mathtt{aB} = (l_1{\cdot}l_2).\mathit{get}\ \mathtt{a0b1}. \end{array}$$

As with union, we deal with this implicit flow of information by raising the confidentiality level of the data in the view, annotating the view type with the least label that observes the unambiguous concatenation of the source types.

Formally, we say k *observes* $(S_1{\cdot^!}S_2)$ iff for every $s_1{\cdot}s_2$ and $s_1'{\cdot}s_2' \in S_1{\cdot}S_2$ with $s_1{\cdot}s_2 \sim_k s_1'{\cdot}s_2'$ we have $s_1 {\sim_k} s_1'$ and $s_2 \sim_k s_2'$. We can effectively compute whether a given label observes the unambiguous concatenation of two annotated regular expressions using an elementary construction.

In the reverse direction, the concatenation lens splits the source and view strings in two, applies the *put* components of l_1 and l_2 to the corresponding pieces of each, and concatenates the results. An analogous problem now arises with integrity, so we escalate the label on the source type with the least label that observes the unambiguous concatenation of the view types.

Iteration The next combinator iterates a lens.

$$\frac{\begin{array}{c} l \in S \Longleftrightarrow V \qquad S^{!*} \qquad V^{!*} \\ q = \bigvee \{k \mid k \text{ min obs. } V^{!*}\} \\ p = \bigvee \{k \mid k \text{ min obs. } S^{!*}\} \end{array}}{l^* \in (S^*){:}\,q \Longleftrightarrow (V^*){:}\,p}$$

$$\begin{array}{ll} \mathit{get}\ (s_1 \cdots s_n) & = (l.\mathit{get}\ s_1) \cdots (l.\mathit{get}\ s_n) \\ \mathit{put}\ (v_1 \cdots v_n)\ (s_1 \cdots s_m) & = s_1' \cdots s_n' \\ \quad \text{where } s_i' = \begin{cases} l.\mathit{put}\ v_i\ s_i & i \in \{1, ..., \min(m,n)\} \\ l.\mathit{create}\ v_i & i \in \{m+1, ..., n\} \end{cases} & \\ \mathit{create}\ (v_1 \cdots v_n) & = (l.\mathit{create}\ v_1) \cdots (l.\mathit{create}\ v_n) \end{array}$$

As with union and concatenation, we need to escalate the confidentiality label on the view side and the integrity label on the source side. To see why, consider the following lens:

$$l \quad \triangleq \quad \mathtt{A}{:}\mathsf{S} \leftrightarrow \mathtt{B}{:}\mathsf{P} \quad \in \quad \mathtt{A}{:}\mathsf{S} \Longleftrightarrow \mathtt{B}{:}\mathsf{P}$$

It is not the case that

$$l^* \in (\mathtt{A}{:}\mathsf{S})^* \Longleftrightarrow (\mathtt{B}{:}\mathsf{P})^*,$$

as demonstrated by the following counterexample to GETNOLEAK:

$$\begin{array}{ll} & \mathtt{AAA} \sim_{\mathsf{P}} \mathtt{AA} \\ \text{but} & l^*.\mathit{get}\ \mathtt{AAA} = \mathtt{BBB} \not\sim_{\mathsf{P}} \mathtt{BB} = l^*.\mathit{get}\ \mathtt{BB}. \end{array}$$

The problem is that *get* leaks the length of the source string, which is secret. Thus, we need to escalate the confidentiality label on the view type by the least label observing the unambiguous iterability of the source type.

Likewise, if we consider integrity, it is not the case that the iteration of

$$l \triangleq \mathtt{[0\text{-}9]}{:}\mathsf{E} \leftrightarrow \mathtt{A}{:}\mathsf{T} \in \mathtt{[0\text{-}9]}{:}\mathsf{E} \Longleftrightarrow \mathtt{A}{:}\mathsf{T}$$

has type

$$l^* \in (\mathtt{[0\text{-}9]}{:}\mathsf{E})^* \Longleftrightarrow (\mathtt{A}{:}\mathsf{T})^*,$$

as demonstrated by the following counterexample to GETPUT:

$$\begin{array}{ll} & \mathtt{A} \approx_{\mathsf{E}} \mathtt{AAA} = l^*.\mathit{get}\ \mathtt{123} \\ \text{but} & l^*.\mathit{put}\ \mathtt{A}\ \mathtt{123} = \mathtt{1} \not\approx_{\mathsf{E}} \mathtt{123}. \end{array}$$

Here the problem is that the update shortens the length of the view, which causes the iteration operator to discard endorsed data in the source. Thus, we need to escalate the integrity label by the join of the minimal label that observes the unambiguous iterability of V. These labels can be computed from annotated regular expressions using elementary constructions.

Sequential Composition The next operator composes lenses sequentially.

$$\frac{l_1 \in S \Longleftrightarrow T \qquad l_2 \in T \Longleftrightarrow V}{l_1; l_2 \in S \Longleftrightarrow V}$$

$$\begin{array}{ll} \mathit{get}\ s & = l_2.\mathit{get}\ (l_1.\mathit{get}\ s) \\ \mathit{put}\ v\ s & = l_1.\mathit{put}\ (l_2.\mathit{put}\ v\ (l_1.\mathit{get}\ s))\ s \\ \mathit{create}\ v & = l_1.\mathit{create}\ (l_2.\mathit{create}\ v) \end{array}$$

In the forward direction it applies the *get* of l_1 followed by the *get* of l_2, and in the reverse direction it applies the *put* of l_2 followed by the *put* of l_1 (using l_1's get to generate a T to use as the source argument for l_2's *put*). The typing rule requires that the view type of the first lens and the source type of the second must be identical.

Filter The *filter* lens takes as arguments two well-formed annotated regular expressions E and F, which must be disjoint, and produces a lens that, in the *get* direction, takes a string of Es and Fs and filters away the Fs, and, in the *put* direction, weaves together an updated view with the original source, propagating the changes made to the list of Es in the view and restoring the Fs from the source by position. It is tempting to define *filter* as $(copy\ E \mid del\ F)^*$ but our type system disallows this grouping of combinators—the view type of lens being iterated is not unambiguously iterable (as it contains ϵ); moreover, its *put* function does not have the same behavior as *unfilter*, which always restores all the Fs from the source. But we can easily define a primitive lens that has the following filter and unfilter functions as its *get* and *put* components.

```
let rec filter $E$ $xs$ = match $xs$ with
  | $\epsilon \rightarrow \epsilon$
  | $x{\cdot}xs' \rightarrow$ if $x \in E$ then $x{\cdot}$(filter $E$ $xs'$) else (filter $E$ $xs'$)

let rec unfilter $F$ $es$ $xs$ = match $es, xs$ with
  | $\epsilon, \_ \rightarrow$ filter $F$ $xs$
  | $e{\cdot}es', x{\cdot}xs' \rightarrow$
      if $x \in F$ then $x{\cdot}$(unfilter $es$ $xs'$)
      else $e{\cdot}$(unfilter $es'$ $xs'$)
```

The definition of the *filter* lens is as follows.

$$\frac{\begin{array}{c} E, F \text{ well-formed} \qquad E \cap F = \emptyset \qquad (E \mid F)^{!*} \\ q = \bigvee \{k \mid k \text{ min obs. } E^{!*}\} \\ p \sqsupseteq \bigvee\{k \mid k \text{ observes } E{\cdot}^{!}F \text{ and } F{\cdot}^{!}E\} \end{array}}{\textit{filter}\ E\ F \in (E{:}q \mid F{:}p)^* \Longleftrightarrow E^*}$$

$$\begin{array}{ll} \textit{get}\ (s_1 \cdots s_n) & = \text{filter}\ E\ (s_1 \cdots s_n) \\ \textit{put}\ (v_1 \cdots v_n)\ (s_1 \cdots s_m) & = \text{unfilter}\ F\ (v_1 \cdots v_n)\ (s_1 \cdots s_m) \\ \textit{create}\ (v_1 \cdots v_n) & = (v_1 \cdots v_n) \end{array}$$

The typing rule for *filter* captures the fact that none of the Fs are leaked to the view, and so the Fs in the source can be assigned any confidentiality label (that observes the unambiguous concatenation of Es and Fs). Since observers with clearance lower than p cannot distinguish source strings that differ only in the Fs, it is simple to show GETNOLEAK: two source strings are related by $\sim_{\mathsf{P}}$ exactly when their filterings—i.e., the views computed by *get*—are related by $\sim_{\mathsf{P}}$. In the reverse direction, we need to escalate the integrity label on the Es by the join of the minimal label that observes that E is unambiguously iterable. However, the Fs are restored exactly, so their integrity level does not need to be escalated.

Subsumption Secure lenses also admit a rule of subsumption that allows us to escalate the integrity level on the source and the confidentiality level on the view.

$$\frac{l \in S \Longleftrightarrow V \qquad q \in \mathcal{Q} \qquad p \in \mathcal{P}}{l \in (S{:}q) \Longleftrightarrow (V{:}p)}$$

It may seem silly to escalate labels arbitrarily, but it is occasionally useful—e.g., to make the types agree when forming the sequential composition of two lenses.

6. Dynamic Secure Lenses

Using the static type system to track tainted source data is effective, but conservative—it forces us to label source data as tainted if the *put* function ever produces a tainted result, even if there are many inputs for which it does not. In this section, we explore the idea of augmenting lenses with dynamic tests to check whether *put* can preserve the integrity of the endorsed data in the source for a *particular* view and source. This generalization makes it possible for lenses to make very fine-grained decisions about which views to accept and which to reject, and allows us to assign relaxed types to many of our lens primitives while still retaining strong guarantees about integrity.

At the same time that we extend lenses with these dynamic tests, we also address a subtle interaction between confidentiality and integrity that we have ignored thus far. In the preceding sections, we have assumed that the confidentiality and integrity annotations are completely orthogonal—the semantics of types treats them as independent, and each behavioral law only mentions a single kind of label. However, the protocol for propagating updates to views, in which the owner of the source data tests whether the original and updated views agree on endorsed data, can reveal information—possibly confidential—about the source. In this section, we eliminate the possibility of such leaks by adding a new behavioral law requiring that testing whether a given view can be handled (now using arbitrary dynamic tests) must not leak confidential information. (An analogous fix can be made in the purely static type system described in the preceding section.)

Formally, we let $\mathcal{C} \subseteq \mathcal{P} \times \mathcal{Q}$ be a set of *clearances*. A user with clearance (j, k) is allowed to access data at confidentiality level j and modify data tainted at integrity level k. We extend lenses with a new function

$$l.\textit{safe} \in \mathcal{C} \longrightarrow \mathcal{L}(V) \longrightarrow \mathcal{L}(S) \longrightarrow \mathbb{B}$$

that returns true iff a user with clearance (j, k) can safely *put* a particular view and source back together. We replace the hypothesis that $v \approx_k s$ in the GETPUT law with *safe*, requiring, for all $(j, k) \in \mathcal{C}$ and $s \in S$ and $v \in V$ that

$$\frac{l.\textit{safe}\ (j,k)\ v\ s}{l.\textit{put}\ v\ s \approx_k s} \qquad \text{(GETPUT)}$$

and we revise the protocol for propagating updates to the view accordingly: before allowing the user of a view to invoke *put*, the owner of the source checks that the original and updated views are safe for the user's clearance.

As discussed above, these *safe* functions, which are arbitrary functions, can reveal information about source data. We therefore add a new law stipulating that *safe* must not reveal confidential data, formulated as a non-interference property for every $(j,k) \in \mathcal{C}$, every $s, s' \in S$, and $v, v' \in V$:

$$\frac{v \sim_j v' \qquad s \sim_j s'}{l.\mathit{safe}\ (j,k)\ v\ s = l.\mathit{safe}\ (j,k)\ v'\ s'}\ (\text{SAFENOLEAK})$$

For technical reasons—to prove that the *safe* component of the sequential composition operator, which is defined in terms of the *put* function of one of its sublenses, satisfies SAFENOLEAK—we also need a law stipulating that the *put* function must be non-interfering for all $(j,k) \in \mathcal{C}$ and for all $s, s' \in S$ and $v, v' \in V$:

$$\frac{\begin{array}{cc} v \sim_j v' & s \sim_j s' \\ l.\mathit{safe}\ (j,k)\ v\ s & l.\mathit{safe}\ (j,k)\ v\ s' \end{array}}{l.\mathit{put}\ v\ s \sim_j l.\mathit{put}\ v'\ s'}\ (\text{PUTNOLEAK})$$

With these refinements, we can now present revised versions of each of our secure string lens combinators.

For *copy* the *safe* function checks whether the new view and original source agree on k-integrity data.

$$\boxed{\begin{array}{c} \dfrac{E \text{ well-formed} \qquad \forall\ (j,k) \in \mathcal{C}.\ \sim_j \subseteq \approx_k}{\mathit{copy}\ E \in E \Longleftrightarrow E} \\ \\ \mathit{safe}\ (j,k)\ v\ s = v \approx_k s \end{array}}$$

To ensure that *safe* does not leak information, we add a hypothesis to the typing rule requiring that $\sim_j$ must refine $\approx_k$ for every $(j,k) \in \mathcal{C}$. This condition captures the essential interaction between the confidentiality and integrity lattices.

For *const*, the view type is a singleton, so there is only one possible update—a no-op. Hence, we choose a *safe* function that always returns true.

$$\boxed{\begin{array}{c} \dfrac{E, F \text{ well-formed} \qquad |F| = 1 \qquad d \in E}{\mathit{const}\ E\ F\ d \in E \Longleftrightarrow F} \\ \\ \mathit{safe}\ (j,k)\ v\ s = \mathit{true} \end{array}}$$

For the concatenation lens, we choose a *safe* function that tests if the unique substrings of the source and view are safe for l_1 and l_2. It also checks whether j observes the unambiguous concatenation of the source and view types—this is needed to prove PUTNOLEAK and SAFENOLEAK.

$$\boxed{\begin{array}{l} \dfrac{\begin{array}{c} l_1 \in S_1 \Longleftrightarrow V_1 \quad S_1 \cdot^! S_2 \\ l_2 \in S_2 \Longleftrightarrow V_2 \quad V_1 \cdot^! S_2 \\ p = \bigvee\{k \mid k \text{ min obs. } S_1 \cdot^! S_2\} \end{array}}{l_1 \cdot l_2 \ \in\ (S_1 \cdot S_2) \Longleftrightarrow (V_1 \cdot V_2){:}\,p} \\ \\ \mathit{safe}\ (j,k)\ v_1.v_2\ s_1.s_2 = \\ \quad j \text{ observes } S_1 \cdot^! S_2 \text{ and } V_1 \cdot^! V_2 \\ \quad \wedge\, l_1.\mathit{safe}\ (j,k)\ v_1\ s_1 \wedge l_2.safe\ (j,k)\ v_2\ s_2) \end{array}}$$

For the union lens, the *safe* function tests whether the source and view can be processed by the same sublens. (Additionally, because *safe* can be used to determine whether the source came from S_1 or S_2, it only returns true if j observes their disjointness and if V_1 and V_2 agree in their intersection.)

$$\boxed{\begin{array}{l} \dfrac{\begin{array}{c} (S_1 \cap S_2) = \emptyset \\ l_1 \ \in\ S_1 \Longleftrightarrow V_1 \qquad l_2 \ \in\ S_2 \Longleftrightarrow V_2 \\ p = \bigvee\{k \mid k \text{ min obs. } (S_1 \cap S_2) = \emptyset\} \end{array}}{l_1 \mid l_2 \ \in\ (S_1 \mid S_2) \Longleftrightarrow (V_1 \mid V_2){:}\,p} \\ \\ \mathit{safe}\ (j,k)\ v\ s = \\ \quad j \text{ observes } (S_1 \cap S_2) = \emptyset \text{ and } V_1\ \&\ V_2 \text{ agree} \\ \quad \bigwedge \begin{cases} l_1.\mathit{safe}\ (j,k)\ v\ s & \text{if } v \in V_1 \wedge s \in S_1 \\ l_2.\mathit{safe}\ (j,k)\ v\ s & \text{if } v \in V_2 \wedge s \in S_2 \\ \mathit{false} & \text{otherwise} \end{cases} \end{array}}$$

For the iteration lens, *safe* checks that the view is the same length as the one generated from the source. Because *safe* can be used to determine the length of the source, we require that j observe the unambiguous concatenation of S and V (which implies that j can distinguish strings of different lengths).

$$\boxed{\begin{array}{l} \dfrac{\begin{array}{c} l \in S \Longleftrightarrow V \\ p = \bigvee\ \{k \mid k \text{ min obs. } S^{!*}\} \end{array}}{l^* \in S^* \Longleftrightarrow (V^*){:}\,p} \\ \\ \mathit{safe}\ (j,k)\ (v_1 \cdots v_n)\ (s_1 \cdots s_m) = \\ \quad j \text{ observes } S^{!*} \text{ and } V^{!*} \\ \quad \wedge\, n = m\ \wedge\ l.\mathit{safe}\ (j,k)\ v_i\ s_i \text{ for } i \in \{1, ..., n\} \end{array}}$$

For sequential composition, the *safe* function requires the conditions implied by l_1's *safe* function on the intermediate view computed by l_2's *put* on the view and the original source.

$$\boxed{\begin{array}{l} \dfrac{l_1 \in S \Longleftrightarrow T \qquad l_2 \in T \Longleftrightarrow V}{l_1; l_2 \in S \Longleftrightarrow V} \\ \\ \mathit{safe}\ (j,k)\ s\ v = l_1.\mathit{safe}\ (j,k)\ (l_2.\mathit{put}\ v\ (l_1.\mathit{get}\ s))\ s \end{array}}$$

Note that the composition operator is the reason we need the PUTNOLEAK law and that SAFENOLEAK needs to require that *safe* be non-interfering in both its source and view arguments (rather than just its source argument). We could relax these conditions by only requiring PUTNOLEAK of lenses used as the second argument to a composition operator and the full version of SAFENOLEAK of lenses used as the first argument. This would give us yet more flexibility in designing *safe* functions (at the cost of complicating the type system since we would need to track several different kinds of lens types). We defer this extension to future work.

Finally, the *safe* function for the *filter* lens checks that the new view and filtered source agree on k-endorsed data. Additionally, to ensure that *safe* does not leak information about the source, *safe* also checks that j observes the way the way that Es and Fs are split in the source, as well as the unambiguous iterability of E.

$$\frac{\begin{array}{c} E, F \text{ well-formed} \qquad E \cap F = \emptyset \qquad (E \mid F)^{!*} \\ p \sqsupseteq \bigvee\{k \mid k \text{ observes } E \cdot^{!} F \text{ and } F \cdot^{!} E\} \\ \forall\ (j,k) \in \mathcal{C}.\ \sim_j^E \subseteq \approx_k^E \end{array}}{\textit{filter } E\ F \in (E \mid F{:}p)^* \Longleftrightarrow E^*}$$

$$\begin{array}{l} \textit{safe } (j,k)\ (v_1 \cdots v_n)\ (s_1 \cdots s_m) = \\ \quad j \text{ observes } E \cdot^{!} F \text{ and } F \cdot^{!} E \quad \wedge\ j \text{ and } k \text{ observe } E^{!*} \\ \quad \wedge\ (v_1 \cdots v_n) \approx_k (\mathsf{filter}\ E\ (s_1 \cdots s_m)) \end{array}$$

The revised lens definitions in this section illustrate how dynamic tests can be incorporated into the secure lens framework, providing fine-grained mechanisms for updating security views and relaxed types for many of our secure string lens combinators. However, they represent just one point in a large design space. We can imagine wanting to equip lenses with several different *safe* functions—e.g., some accepting more views but offering weaker guarantees about the integrity of source data, and others that accept fewer views but offer correspondingly stronger guarantees. We plan to investigate the tradeoffs along these axes in the future.

7. Related Work

This paper builds on our previous work on lenses [19], [8], [7], [21]. A number of other bidirectional languages have also been proposed [37], [41], [26], [9], [24], [35], [17], [23], [30], [28]. The original lens paper [19] includes an extensive survey of the relationship between lenses and approaches to the view update problem in the database literature.

Views have long been used to enforce security boundaries in relational database systems. They were first proposed as a security mechanism for XML data by Stoica and Farkas [38] and were later studied extensively by Fan and his colleagues in a series of papers [14], [15], [16]. The key difference between previous work on security views and the framework proposed in this paper, of course, is support for updates. Additionally, previous systems do not provide a way to formally characterize the data kept confidential by the view—the query that defines the view essentially *is* the privacy policy. Lastly, views in previous systems have typically been virtual, while the views constructed using lenses are materialized. Fan [14] has argued that materializing views is not practical, because many different security views are often needed when policies are complex. We find this argument compelling in the traditional database setting, where data sources are typically very large, but believe that there are also many applications where building materialized security views will be practical. Moreover, in at least some applications, views *must* be materialized—e.g., in the Intellipedia system discussed in the introduction, the regraded documents need to be sent over the network and displayed in a web browser.

The idea of using static analyses to track flows of information in programs was originally proposed by Denning and Denning [13] and has since been applied in a variety of languages, including Jif [32], a secure variant of Java, and FlowCaml [33], a secure variant of OCaml. The excellent survey article by Sabelfeld and Myers [34] gives a general overview of the entire area and provides numerous citations.

Rather less work has focused on applying information-flow analyses to data processing languages. The developers of ℂDuce, a functional language for processing XML data, studied an extension of the language where labels corresponding to security levels are propagated dynamically [2]. Foster, Green, and Tannen proposed a mechanism for ensuring non-interference properties of tree transformations using a semantics that propagates dynamic provenance annotations [18]. The Fable language also propagates security labels dynamically [39], [11]. Fable does not fix a particular semantics for label propagation, but instead provides a general framework that enforces a strict boundary between ordinary program code, which must treat labels opaquely, and security code, which may manipulate labels freely. Thus, it can be used to implement a variety of static and dynamic techniques for tracking information flows in programs. Cheney, Ahmed, and Ucar have introduced a general framework for comparing static and dynamic approaches to many dependency analyses including information flow [10].

Integrity can be treated as a formal dual to confidentiality, as was first noted by Biba [5]. Thus, most of the languages discussed above can also be used to track integrity properties of data. However, as noted by Li, Mao, and Zdancewic [25], information-flow analyses provide weaker guarantees for integrity compared to confidentiality when code is untrusted. Specific mechanisms for tracking integrity have also been included in a variety of languages: Perl has a simple taint tracking mechanism for data values [42]. Wassermann and Su proposed a more powerful approach based on a dynamic analysis of generated strings that tracks tainted data in PHP scripts [43]. Shankar et al. developed a taint analysis for C code using the `cqual` system. Finally, researchers at IBM have recently implemented a taint analysis tool for Java designed to scale to industrial-size web applications [40].

The Intellipedia example discussed in the introduction is partly inspired by a system for building "tearline" security views of MediaWiki documents developed by Galois Inc.

8. Future Work

We are adding the features described in this paper to the Boomerang implementation. We can already use Boomerang to develop lens programs that define updatable security views, since the functional components of the secure lens combinators and their basic lens versions are identical.

However, the type system in the current implementation only tracks regular types. To bring it up to speed, we need an implementation of annotated regular types, which in turn requires implementing a library for representing rational functions and deciding properties such as equivalence and the various observability conditions in the typing rules.

We have presented both a static type system for secure lenses and an approach using dynamic very well behaved lenses. We would like to explore connections with other dynamic approaches—e.g., languages that propagate dynamic labels [44], [36] and provenance metadata. We hope that these languages will suggest mechanisms for enforcing security properties at finer levels of granularity than our current, static, approach can track. We would also like to explore declassification operators [31], quantitative measures of information flow [27], and formal notions of privacy [29].

We would also like to further explore some of the alternative semantics for annotated regular types that we mentioned in Section 3. We chose to work with the erasing semantics in this paper because it seems simplest, but we expect there will be applications where the redacting semantics is a more natural fit. We are also interested in relations for integrity capturing the notion that a modified string "extends" another.

Finally, we would like to develop security-annotated type systems and secure lenses in other settings besides strings—e.g., trees, relations, and graphs. For relations, the bidirectional language proposed by Bohannon et al. [8] should be a good starting point.

Acknowledgments We are grateful to Mike Hicks, Andrew Myers, Jeff Vaughan, and Geoff Washburn for stimulating discussions; to Philip Wadler for suggesting that we investigate dynamic very well behavedness; and to Davi Barbosa, Kathleen Fisher, and the anonymous referees for helpful comments on an earlier draft. Our work is supported by the National Science Foundation under grants IIS-0534592 *Linguistic Foundations for XML View Update*, and CT-0716469 *Manifest Security*.

References

[1] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, December 1981.

[2] Vronique Benzaken, Marwan Burelle, and Giuseppe Castagna. Information flow security for XML transformations. In *Advances in Computing Science: Programming Languages and Distributed Computation (ASIAN), Mumbai, India*, volume 2896 of *Lecture Notes in Computer Science*, pages 33–53, 2003.

[3] Jean Berstel. *Transductions and Context-Free Languages*. Teubner Verlag, 1979.

[4] Jean Berstel, Dominique Perrin, and Christophe Reutenauer. *Codes and Automata*. Cambridge University Press, 2009. To appear. Manuscript available from `http://www-igm.univ-mlv.fr/~berstel/LivreCodes/`.

[5] Kenneth J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR 76-372, The MITRE Corporation, 1977.

[6] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. *XQuery 1.0: An XML Query Language*. W3C, January 2007. Available from `http://www.w3.org/TR/xquery`.

[7] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, CA*, pages 407–419, January 2008.

[8] Aaron Bohannon, Jeffrey A. Vaughan, and Benjamin C. Pierce. Relational lenses: A language for updateable views. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS), Chicago, TL*, 2006. Extended version available as University of Pennsylvania technical report MS-CIS-05-27.

[9] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 33(4–5):385–406, 2008. Short version in DBPL '05.

[10] James Cheney, Amal Ahmed, and Umut A. Acar. Provenance as dependency analysis. In *Symposium on Database Programming Languages (DBPL), Vienna, Austria*, pages 138–152, 2007.

[11] Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Combining provenance and security policies in a web-based document management system. In *On-line Proceedings of the Workshop on Principles of Provenance (PrOPr), Edinburgh, Scotland*, November 2007. `http://homepages.inf.ed.ac.uk/jcheney/propr/`.

[12] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 7(3):381–416, September 1982.

[13] Dorothy E. Denning and Peter J Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[14] Wenfei Fan, Chee Yong Chan, and Minos N. Garofalakis. Secure XML querying with security views. In *ACM SIGMOD International Conference on Management of Data (SIGMOD), Paris, France*, pages 587–598, 2004.

[15] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. SMOQE: A system for providing secure access to XML. In *International Conference on Very Large Data Bases (VLDB), Seoul, Korea*, pages 1227–1230, September 2006.

[16] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Rewriting regular XPath queries on XML views. In *International Conference on Data Engineering (ICDE), Istanbul, Turkey*, pages 666–675, April 2007.

[17] Kathleen Fisher and Robert Gruber. PADS: a domain-specific language for processing ad hoc data. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Chicago, IL*, pages 295–304, 2005.

[18] J. Nathan Foster, Todd J. Green, and Val Tannen. Annotated XML: Queries and provenance. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS), Vancouver, BC*, pages 271–280, June 2008.

[19] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view update problem. *ACM Transactions on Programming Languages and Systems*, 29(3), May 2007.

[20] J. Nathan Foster, Benjamin C. Pierce, and Steve Zdancewic. Updatable security views. Technical Report MS-CIS-09-05, Department of Computer & Information Science, University of Pennsylvania, 2009.

[21] J. Nathan Foster, Alexandre Pilkiewcz, and Benjamin C. Pierce. Quotient lenses. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Victoria, BC*, pages 383–395, September 2008.

[22] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems (TODS)*, 13(4):486–524, 1988.

[23] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1–2), June 2008.

[24] Shinya Kawanaka and Haruo Hosoya. bixid: a bidirectional transformation language for XML. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Portland, Oregon*, pages 201–214, 2006.

[25] Peng Li, Yun Mao, and Steve Zdancewic. Information Integrity Policies. In *Proceedings of the First Workshop on Formal Aspects in Security and Trust (FAST)*, Pisa, Italy, September 2003.

[26] David Lutterkort. Augeas–A configuration API. In *Linux Symposium, Ottawa, ON*, pages 47–56, 2008.

[27] Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Tuscon, AZ*, pages 193–205, 2008.

[28] Lambert Meertens. Designing constraint maintainers for user interaction, 1998. Manuscript, available from `ftp://ftp.kestrel.edu/pub/papers/meertens/dcm.ps`.

[29] Gerome Miklau and Dan Suciu. A formal analysis of information disclosure in data exchange. *Journal of Computer and Systems Sciences*, 73(3):507–534, 2007.

[30] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An algebraic approach to bi-directional updating. In *ASIAN Symposium on Programming Languages and Systems (APLAS)*, pages 2–20, November 2004.

[31] Andrew Myers and Barbara Liskov. A decentralized model for information flow control. In *ACM Symposium on Operating Systems Principles (SOSP), Saint Malo, France*, pages 129–142, 1997.

[32] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Antonio, TX*, pages 228–241, 1999.

[33] Franois Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.

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

[35] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *International Workshop Graph-Theoretic Concepts in Computer Science, Herrsching, Germany*, volume 903 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[36] Paritosh Shroff, Scott F. Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *20th IEEE Computer Security Foundations Symposium (CSF)*, pages 203–217, July 2007.

[37] Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS), Nashville, TN*, volume 4735 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2007.

[38] Andrei Stoica and Csilla Farkas. Secure XML views. In *IFIP WG 11.3 International Conference on Data and Applications Security (DBSEC), Cambridge, UK*, pages 133–146, 2002.

[39] Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 369–383, May 2008.

[40] Omer Tripp, Marco Pistoia, Stephen Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective taint analysis of web applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Dublin, Ireland*, 2009. To appear.

[41] Janis Voigtländer. Bidirectionalization for free! In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Savannah, GA*, pages 165–176, January 2009.

[42] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl (3rd Edition)*. O'Reilly, July 2000.

[43] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), San Diego, CA*, pages 32–41, 2007.

[44] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2–3):67–84, March 2007.