Restricted Delegation and Revocation in Language-Based Security (Position Paper)

D. Hassan
National Telecommunication Institute, Egypt
doaa@nti.sci.eg

M.R. Mousavi
TU/Eindhoven, The Netherlands
m.r.mousavi@tue.nl

M.A. Reniers
TU/Eindhoven, The Netherlands
m.a.reniers@tue.nl

ABSTRACT

In this paper, we introduce a notion of restricted revocable delegation and study its consequences in language-based security. In particular, we add this notion by means of *delegate* and *revoke* commands to a simple imperative programming language. We then define an operational semantics for our programming language, in the Natural Semantics style of Gilles Kahn. We briefly discuss our initial ideas about the security properties of the semantics, which are extensions of existing variations of the renown non-interference property, e.g., in the context of delimited information release.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory

Keywords

Language-Based Security, Operational Semantics, Secure Information Flow, Delegation

1. INTRODUCTION

Security-typed programming languages [18] provide a means to enforce end-to-end information flow security. In these languages, data types are annotated with security labels in order to identify the confidentiality and/or integrity policy for each data element. Such policies specify which principals or entities are allowed to infer or modify the value of data respectively. The specification of principals in security-typed programming languages allows modeling different roles in the application program with different security concerns such as users, groups and processes. In practical applications, however, it is often handy to temporarily modify these policies, i.e., to permit a temporary information flow to a certain principal and renounce a prior permission. We refer to the first phenomenon as delegation and to the second as revocation. Next, we mention two practical examples of this phenomenon. We use the first example as our running example throughout this paper to illustrate different concepts and issues and their formalization.

Example 1 Consider a clinical system in which the physician treating a patient can access the medical history of the patient. Suppose that, in a particular case, the physician would like to delegate her authority to access the medical history to a colleague who can provide a second opinion and add it to the document. However, the physician in charge as well as the patient do not want to enable the second physician to delegate this authority further to a third person. This scenario illustrates the essence of restricted delegation; in practice the permitted chain/tree of delegation may of course be much more complicated than the simple chain sketched above.

Example 2 As a second example, consider an online bookshop which receives the authorization from a customer to charge her credit card. Hence, during the transaction the customer delegates the authority of accessing her credit card information to the bookshop, with a further permission to authorize the transaction with (delegate the read and write permission to) the bank. However, upon finalizing the transaction, the client revokes this delegation. After revocation, the bookshop is not allowed to keep hold of and access client's credit card information. This is an example of a revocation.

Related work..

A general form of delegation is already present in some security-typed programming languages such as Jif [8, 15]. Using delegation in Jif a principal p can *act for* another principal q, i.e., any action taken by principal q is also authorized for principal p. However, this feature does not support any of the features mentioned above, i.e., restricted and revocable delegation. These two issues have been studied in other areas such as access control systems [2] and workflow management [22]. However, there exists only few attempts to accommodate these concepts in language-based security [4, 12, 6, 7]. Our work is inspired by and improves upon [6]. In [6] delegation within a certain scope is introduced and is used to ensure confidentiality. Our work extends this to guarantee integrity and adds the aspect of delegation chaining to it. In [7], a dynamic model of information flow is presented by means of conditional flow rules. We expect that our delegation and revocation chains can be coded in the approach of [7] by using auxiliary variables and predicates,

```
Body      ::=(VarDecl | Statement)*
VarDecl   ::=Var ":" SecType ":=" Val ";"
SecType   ::="("Type"," Label")"
Type      ::="Bool" | "Int" | ...
Label     ::=Policy*
Policy    ::=Principal [":" (["!"]Principal)*]
Principal ::="p" | "q" | "r" | ...
Statement::=Var ":=" Exp ";" |
        if Exp then Body else Body  fi|
        while Exp do Body od |
        Principal delegates Var("(*)" |
        "(!)" | "(*,!)") Chain ";" |
        Principal revokes Var ("(*)" | "(!)" | "(*,!)") Chain ";"
Exp       ::=Var | Cons | Exp op Exp
Chain     ::=Principal | Principal "→" Chain
```

Figure 1: Syntax of An Imperative Programming Language.

which are used in the conditional flow rules and are (re)set in case of delegation (revocation). However, this coding is very cumbersome and harms the readability of the code, and moreover, the information leaked through a delegation goes beyond the control of the grantor and can be accessed after the revocation.

It has already been observed by several researchers, e.g., in [6, 13, 17], that the original notion of non-interference [9] is too "static" to track the information flow in the presence of dynamic flow constructs such as those for declassification [17]. In this paper, we briefly present our initial ideas about the security properties suitable for the setting with delegation and revocation.

Structure of the paper..

The rest of this paper is structured as follows. In Section 2, we define the syntax and briefly describe the informal semantics of a simple security-typed imperative programming language which includes restricted delegation and revocation constructs. In Section 3, we formalize the semantics of our language in the Natural Semantics style of Gilles Kahn [10]. Section 4 studies security properties, à la non-interference [9], in the presence of delegation and revocation. Section 5 concludes the paper and presents some directions for future research.

2. SYNTAX AND INFORMAL SEMANTICS

The syntax of our simple imperative programming language is given in Figure 1.

Next, we explain the intuitive meaning of the constructs presented in Figure 1. A program is a sequential piece of code, whose body consists of a number of variable declarations and statements. Variables have a security type, which is an ordinary data type augmented with a security label. A security label consists of zero or more security policies to enforce confidentiality and integrity respectively. A security policy of the form "p : q ; ! r" specifies that principal p is an owner of the declared variable, while principals q and r can, respectively, read from and write to the information stored by p in this variable. Multiple policies may define different owners for the same variable, with different readers and writers for each owner, where the reader can read the owner data, while the writer can edit her data. The read and write permission associated with each owner refers to the data values stored by the owner(s) in that variable. (Intuitively we take multiple policies to collectively define a set of owners, readers and writers; if desired, one could modify our proposed semantics to allow for readers and writers defined in each policy to read from and write on the values assigned by the owner in the same policy.) Note that this initial policy can be changed later in the course of program execution by means of delegation and revocation. A statement can be an assignment, a conditional or a loop statement. Moreover, the owner principal of a variable can delegate read/write authority over the variable to a chain of principles. The initial element of the chain is the one receiving the authority but may delegate it further to the other elements of the chain depending on the privileges given to it by the grantor principal. The dual to delegation is revocation and specifies that a certain delegation (sub-)chain is not valid anymore and moreover, no obtained information in the course of delegation may be leaked to the principals whose authorities are revoked. In this paper, we assume that delegation is not necessarily transitive, i.e., by specifying a chain, the grantor does also specify the order in which delegation may happen. A transitive delegation operator can then be defined as a syntactic sugar. Along the same line, we define revocation of a sub-chain to result in revoking all the chains following this sub-chain. Again, an alternative revocation statement can be defined where the chain is "patched" by concatenating the rest of the chain to its allowed pre-fix. For simplicity, we assume that each principal is mentioned only once in each delegation chain.

To illustrate our syntax, we specify the informal explanation of Example 1 in our syntax.

Example 3 The following program captures the scenario described in Example 1 using the syntax given in Figure 1.

// *patient* visits *phys1*
history : (String, *patient*) := SQLQuery;
patient **delegates** history(*,!) *phys1*;
obs1 : (String, *phys1 patient*) := Observation;
history := history + obs1;
// *phys1* introduces *phys2* to *patient*
patient **delegates** history(*,!) *phys1* → *phys2*;
phys1 **delegates** history(*,!) *phys2*
obs2 : (String, *phys2 patient*) := NewObs;
// *phys2* makes some observations
history := history + obs2;
// *phys2* takes a leave and delegates her tasks to *phys3*
phys2 **delegates** history(*,!) *phys3* ; // *phys2* delegates the patient's history to *phys3* without any permission from *patient* : error

3. SEMANTICS

3.1 Introduction

We aim for providing an operational semantics for the programming language. We go for a Natural Semantics [10], because it allows us to more naturally deal with the notion of scope and its effect of information flow. Using a bit more of book-keeping one could adapt the given semantics to the small-step setting.

As stated before, the semantics is supposed to control information flow in order to prevent information leakage. For the simplest case of (direct information leak), consider an assignment of the form $x := y$ and suppose that the security label of variable x does not incorporate (i.e., is not more restrictive than) the security label of variable y. Then, information stored in y is made available for reading and writing to principals that have been granted access by the owner of variable x.

But information leak might also be indirect and due to the control structure of the program. Consider a statement if $y = 0$ then $x := 0$ else $x := 1$. In this case, the information about variable y is leaked to x via the if-then-else control structure.

A more intricate form of disallowed information leakage arises from delegation and revocation. Assume that x and y have the same original, henceforth called *static*, security label and suppose that due to a delegation by a principal p not mentioned in the static security label of x and y, some information owned by p is leaked to y. After that p revokes its delegation, the assignment $x := y$ leaks information owned by p to x, which should be prevented by the virtue of the revoked delegation.

Each of the above-mentioned three categories of information-leakage is prevented using a mechanism in the presented semantics. For the simple and direct information leakage, static labels are checked in the premises of the assignment rule. For the indirect information leakage due to control structures the status of the *program counter* is taken into account, following the approach of [14]. To keep track of the dynamic nature of delegation and revocation and the information flow caused by temporary delegations, dynamic labels of variables are stored and updated in the course of variable assignments. Note that at this point, we have to depart from the more static nature of labels in the decentralized label model of [16].

The following types of information are relevant to establish the execution of the program and to track and control the information flow.

1. The *program counter* indicates which principals have influenced the control-flow of the program so far (by means of the conditions of if-then-else and while-do constructs). This is a well-known approach in modeling and preventing undesired implicit information flows [14].

2. The *values of the program variables* are relevant since these are needed to compute values of conditions in guards of if-then-else and while-do constructs and for determining the new values for variables in assignments.

3. The *static label* is the label of the variable from the variable declaration; this label defines which principals are initially allowed to write to and read from a variable.

4. The *dynamic label* captures the label of the variable as it changed during the execution of the program due to delegation and revocation statements.

5. The *information flow* is the set of principals that have contributed to the current value of a variable. This information is needed to prevent information flow from granting principals to the delegates after the delegation is revoked. By keeping the information flow explicit, we can record the influence of principals on variables and thus prevent further use of information flow that was made possible due to a (by now revoked) delegation.

3.2 Configuration (Operational State)

The configurations that we will consider for the structured operational semantics are tuples of the following type: [1]

$$\begin{array}{lcl} \mathit{Config} & = & \mathit{Body} \times \mathit{Environment} \\ \mathit{Environment} & = & \mathit{Valuation} \times \mathit{StaticLabel} \times \\ & & \quad \mathit{DynamicLabel} \times \mathit{Flow} \end{array}$$

The set *Body* describes the remaining part of the body of the program that still needs to be executed. For the language described in this paper this is a (possibly empty) list of variable declarations and statements, as defined by our BNF syntax. The set *Environment* defines for each variable its valuation and the information regarding its allowed and its actual information flow. A *valuation* $v : \mathit{Var} \mapsto \mathit{Val}$ is a partial function from variable names to values. For simplicity, we do not care about the types of the values and assume that they are collectively present in *Val*. We use this partial function in such a way that the variables for which the valuation is defined are declared (already) and those that are not in the domain of the valuation are not declared (yet).

A static label is represented by a partial mapping from variables to a label. A label is a triple of a set of owners, a set of writers and a set readers. Thus

$$\mathit{Label} = \mathcal{P}(\mathit{Principal}) \times \mathcal{P}(\mathit{Principal}) \times \mathcal{P}(\mathit{Principal})$$

This means that the static label is nicely captured by

$$\mathit{StaticLabel} = \mathit{Var} \nrightarrow \mathit{Label}$$

Given a static label s and a variable x, assume that $s(x) = (o, w, r)$; we write $\mathit{owner}(s(x))$ for o, $\mathit{writer}(s(x))$ for $o \cup w$, and $\mathit{reader}(s(x))$ for $o \cup r$, respectively. Intuitively, $\mathit{writer}(s(x))$ and $\mathit{reader}(s(x))$, respectively, refer to the sets of statically authorized writers and readers of x. Here, we follow the decentralized label model if [16] for the interpretation of confidentiality of integrity labels; see [21] for an alternative interpretation of integrity, which could as well be adopted in our setting.

The *dynamic label* defines for each variable two sets of delegation chains of principals: the first one for recording the write delegation chains and the second one for the reader chains. Each chain is a sequence of principal augmented with a natural number indicating the index of the present actual delegation. For each variable, a delegation chain specifies possible delegations in the future; the principals appearing on and before the index of the write and read delegation chains are those that can currently, respectively, write and read the variable in hand due to delegation.

[1] We use a **Z**-like notation [20] for mathematical expressions; in particular, $\nrightarrow$ stands for partial function, $\mathcal{P}$ stands for power set, *Seq* for a sequence, $\frown$ for sequence concatenation, and $\oplus$ for function update (i.e., re-defining the value for an element in the domain, or introducing a value for an element not in the domain). A sequence of elements is a mapping from natural numbers to the set of elements. Natural numbers (and hence the indices of sequences) start from 0.

$$\begin{array}{lcl} \mathit{DynamicLabel} & = & \mathit{Var} \mathrel{+\!\!\!\rightarrow} (\mathcal{P}(\mathit{Chain}) \times \mathcal{P}(\mathit{Chain})) \\ \mathit{Chain} & = & (\mathit{Seq\ Principal}) \times \mathbb{N} \end{array}$$

For a dynamic label d, we write $d_w : \mathit{Var} \mathrel{+\!\!\!\rightarrow} \mathcal{P}(\mathit{Chain})$ to denote the write delegation chains, and d_r (of the same type) to denote the read chains. For a variable x, we write $\mathit{current}(d_r(x))$ (respectively, $\mathit{current}(d_w(x))$) to denote actual writers (respectively, readers), i.e., the principals appearing on and before the index of the corresponding chain specified by the dynamic label. The tails of such chains specify the potential writers and readers which may receive the right to modify or access this variable through the execution of delegation statements.

An alternative semantic construction could only consider delegation sequences only; then, each time a delegation takes place a suffix of the chain in which delegation has taken place is added to the set of sequences. Without revocation, this can lead to a much cleaner semantics, but in the presence of revocation, one should record "where the current delegation has come from". Otherwise, one can revoke a delegation which could have been caused by her but in fact has come into being by another principal. This is efficiently achieved by the above-given semantic construction, which keeps the delegation structure, and only shifts it index forth, each time a delegation takes place.

Finally, the last component of a configuration is *Flow*, which specifies, for each variable, the set of principal which influenced it current value:

$$\mathit{Flow} = \mathit{Var} \mathrel{+\!\!\!\rightarrow} \mathcal{P}(\mathit{Principal})$$

The program counter $PC = \mathcal{P}(\mathit{Principal})$ is the set of principals which have influenced the control flow of the program thus far. The transition relation defined by the operational semantic rules given below is of the following type:

$$PC \vdash \mathit{Body} \times \mathit{Environment} \rightarrow \mathit{Environment}$$

I.e., it defines that under a certain context for program counter, when the body of a program is evaluated under an environment, the resulting environment is the one specified by the target of the transition. By default, the evaluation of a program P starts from the following configuration: $\emptyset \vdash (P, (\emptyset, \emptyset, \emptyset, \emptyset))$, i.e., the PC is not influenced by any principal and valuation, static label, dynamic label and information flow are all partial functions with an empty domain.

3.3 Variable declaration

A variable can de declared just one. This is expressed by checking whether or not the variable that is declared is already in the domain of the valuation. The declared variable is added to this valuation with the initial value assigned to it. Both delegation chains for the declared variable are initially empty and the only principals which have influenced the (initial) value of this variable so far is its set of owners.

$$\frac{\begin{array}{c} v' = v \oplus \{x \mapsto \mathit{val}\} \quad s' = s \oplus \{x \mapsto \mathit{simp}(l)\} \\ d' = d \oplus \{x \mapsto (\emptyset, \emptyset)\} \quad i' = i \oplus \{x \mapsto \mathit{ext\_ow}(l)\} \\ P \vdash \langle S, v', s', d', i' \rangle \rightarrow \langle v'', s'', d'', i'' \rangle \end{array}}{P \vdash \langle x : (T, l) := \mathit{val}; S, v, s, d, i \rangle \rightarrow \langle v'', s'', d'', i'' \rangle}$$

In the above deduction rule the mappings *simp* and *ext_ow* are simple mappings turning a syntactic label to, respectively, a (semantic) triple and a set of principals as defined below. In the following definitions l is a syntactic label, pr and pr' are principals and prs is a list of principals possibly annotated by exclamation marks.

$$\begin{array}{ll} \mathit{simp}(l) & = (\mathit{ext\_ow}(l), \mathit{ext\_re}(l), \mathit{ext\_wr}(l)) \\ & \\ \mathit{ext\_ow}(\epsilon) & = \emptyset \\ \mathit{ext\_ow}(pr :\ prs\ \ l) & = \{pr\} \cup \mathit{ext\_ow}(l) \\ & \\ \mathit{ext\_re}(\epsilon) & = \emptyset \\ \mathit{ext\_re}(pr : \epsilon) & = \emptyset \\ \mathit{ext\_re}(pr :\ pr'\ prs\ l) & = \{pr'\} \cup \mathit{ext\_re}(pr :\ prs) \cup \mathit{ext\_re}(l) \\ \mathit{ext\_re}(pr :\ !pr'\ prs\ l) & = \mathit{ext\_re}(pr : prs) \cup \mathit{ext\_re}(l) \\ & \\ \mathit{ext\_wr}(\epsilon) & = \emptyset \\ \mathit{ext\_wr}(pr : \epsilon) & = \emptyset \\ \mathit{ext\_wr}(pr :\ !pr'\ prs\ l) & = \{pr'\} \cup \mathit{ext\_wr}(pr :\ prs) \cup \mathit{ext\_wr}(l) \\ \mathit{ext\_wr}(pr :\ pr'\ prs\ l) & = \mathit{ext\_wr}(pr :\ prs) \cup \mathit{ext\_wr}(l) \end{array}$$

Example 4 Consider the first statement of the program presented in Example 3, given below.

history : (String, *patient*) := SQLQuery;

Starting from the above statement with the default initial state (i.e., the empty set for the PC and partial functions with empty domain for all components of the environment), we arrive in the following environment, by applying the above-given deduction rule:

$$\begin{array}{l} \mathit{env}_0 {=} (\{\mathit{history} \mapsto \mathit{SQLQuery}\}, \{\mathit{history} \mapsto (\{\mathit{patient}\}, \emptyset, \emptyset)\}, \\ \qquad \{\mathit{history} \mapsto (\emptyset, \emptyset)\}, \{\mathit{history} \mapsto \{\mathit{patient}\}\}) \end{array}$$

3.4 Assignments

In order for an assignment to make a successful operational step, one requires that x and all variables that occur freely in e are defined variables, and

1. either static label and the dynamic information flow influencing e are more restricted than static label of x,
2. or x is allowed to store the information content of e due to a delegation statetment.

The first deduction rule given below realizes the first item and the subsequent rule is dedicated to the second item in the list.

$$\frac{\begin{array}{c} x \in \mathrm{dom}(v) \quad \mathit{vars}(e) \subseteq \mathrm{dom}(v) \\ s(e) \sqsubseteq s(x) \quad P \cup i(e) \sqsubseteq \mathit{writers}(s(e)) \\ i(e) \sqsubseteq \mathit{writers}(s(x)) \\ v' = v \oplus \{x \mapsto \overline{v}(e)\} \quad i' = i \oplus \{x \mapsto i(e) \cup P\} \\ P \vdash \langle S, v', s, d, i' \rangle \rightarrow \langle v'', s', d', i'' \rangle \end{array}}{P \vdash \langle x := e; S, v, s, d, i \rangle \rightarrow \langle v'', s', d', i'' \rangle}$$

The symbol $\sqsubseteq$ should be read as "less restricted than" and is defined by the standard order-theoretic approach to security labels (see, e.g., [16]). We use $\sqsubseteq$ both between labels and sets of principals, but this is understood from the context and should not cause any confusion. Function $\overline{v}$ is a mapping that associates with an expression e (over variables from $\mathrm{dom}(v)$) the value that is obtained by replacing the variables by their values from v. Note that the label and the information flow of an expression e is defined inductively based on its structure, int the standard manner.

The above-given rule first checks whether the static label of the assigned expressions e is more strict than that of the variable x assigned to it. If so, it checks whether the information content of the expression is covered by its static label (no information leakage in the expression due to prior delegations) and finally, whether the principals influencing the expression are allowed to influence the variable.

The other case is where the static labels do not allow for the information flow prescribed by the assignment, but the delegation statements do allow for this. This is captured by the following deduction rule.

$$\frac{\begin{array}{c} x \in \mathrm{dom}(v) \quad \mathit{vars}(e) \subseteq \mathrm{dom}(v) \quad s(e) \not\sqsubseteq s(x) \\ \mathit{readers}(s(e)) \sqsubseteq \mathit{current}(d_r(e)) \cup \mathit{readers}(s(x)) \\ \mathit{writers}(s(x)) \cup \mathit{current}(d_w(x)) \sqsubseteq P \cup i(e) \\ v' = v \oplus \{x \mapsto \overline{v}(e)\} \quad i' = i \oplus \{x \mapsto i(e) \cup P\} \\ P \vdash \langle S, v', s, d, i' \rangle \to \langle v'', s', d', i'' \rangle \end{array}}{P \vdash \langle x := e; S, v, s, d, i \rangle \to \langle v'', s', d', i'' \rangle}$$

In the above deduction rule, the premise $\mathit{readers}(s(e)) \sqsubseteq \mathit{current}(d_r(e)) \cup \mathit{readers}(s(x))$ checks whether the set of dynamic actual readers (i.e., currently delegated readers) of e and static readers of x are large enough to contain all static readers of e. In other words, we check whether the information leak caused by this assignment is justified by the combination of static policies and delegation statements executed so far. Similarly, $\mathit{writers}(s(x)) \cup \mathit{current}(d_w(x)) \sqsubseteq P \cup i(e)$, specifies that those principals that are about to modify x through the assignment, are among those authorized to write on x by its static label or by the delegation statements executed hitherto.

3.5 If-then-else statements

The semantics of if-then-else statements is presented by the following two deduction rules, catering for the "then" and the "else" cases, respectively. Note in both cases that the program counter in the body of the conditional statement is influenced by the principals affecting its condition; hence, within the scope of if-then-else, the program counter is extended with $i(b)$, i.e., the principals which have influenced b so far. This influence does not carry over to the statements executing after the conditional and hence the rest of the program is executed under the original value of the program counter.

$$\frac{\begin{array}{c} \mathit{vars}(b) \subseteq \mathrm{dom}(v) \quad \overline{s}(b) = \mathit{true} \\ P \cup i(b) \vdash \langle S, v, s, d, i \rangle \to \langle v', s', d', i' \rangle \\ P \vdash \langle S'', v', s', d', i' \rangle \to \langle v'', s'', d'', i'' \rangle \end{array}}{P \vdash \langle \text{if } b \text{ then } S \text{ else } S' \text{ fi}; S'', v, s, d, i \rangle \to \langle v'', s'', d'', i'' \rangle}$$

$$\frac{\begin{array}{c} \mathit{vars}(b) \subseteq \mathrm{dom}(v) \quad \overline{s}(b) = \mathit{false} \\ P \cup i(b) \vdash \langle S', v, s, d, i \rangle \to \langle v', s', d', i' \rangle \\ P \vdash \langle S'', v', s', d', i' \rangle \to \langle v'', s'', d'', i'' \rangle \end{array}}{P \vdash \langle \text{if } b \text{ then } S \text{ else } S' \text{ fi}; S'', v, s, d, i \rangle \to \langle v'', s'', d'', i'' \rangle}$$

Note that in order to define a small-step semantics for our programming language, one has to keep track of the current scope of the program counter. Hence, in a small-step semantics, instead of a set of principals, a stack (sequence) of sets of principals should be stored and each time entering (leaving) a scope an updated set pushed into (popped from) the stack.

3.6 While-do statements

The semantics of a while loop is presented by the following two deduction rules. Besides the standard definition, the main twist in the following two definition is that the program counter entities is influenced by the principals involved in defining its boolean condition. This has already been observed in the semantics of the conditional statement. However, the program counter for the continuation of the program is also influenced by the condition of the while loop, because reaching the continuation depends on the termination of the while-loop, and hence on its boolean condition.

$$\frac{\begin{array}{c} \mathit{vars}(b) \subseteq \mathrm{dom}(v) \quad \overline{s}(b) = \mathit{true} \\ P \cup i(b) \vdash \langle S; \text{ while } b \text{ do } S \text{ od}; S', v, s, d, i \rangle \to \langle v', s', d', i' \rangle \end{array}}{P \vdash \langle \text{while } b \text{ do } S \text{ od}; S', v, s, d, i \rangle \to \ \vdash \langle v', s', d', i' \rangle}$$

$$\frac{\begin{array}{c} \mathit{vars}(b) \subseteq \mathrm{dom}(v) \quad \overline{s}(b) = \mathit{false} \\ P \cup i(b) \vdash \langle S', v, s, d, i \rangle \to \langle v', s', d', i' \rangle \end{array}}{P \vdash \langle \text{while } b \text{ do } S \text{ od}; S', v, s, d, i \rangle \to \langle v', s', d', i' \rangle}$$

3.7 Delegation statements

The first three deduction rules of the delegates construct allow for adding a delegation chain by the owner of the variable (for reading, writing or both reading and writing the variable, respectively). Due to space restrictions, we only give the deduction rule for the delegation of reading; the other two rules are almost identical.

$$\frac{\begin{array}{c} x \in \mathrm{dom}(v) \quad d' = d \oplus \{x \mapsto (d_w(x), d_r(x) \cup \{(c, 0)\})\} \\ p \in \mathit{owner}(s(x)) \quad P \vdash \langle S, v, s, d', i \rangle \to \langle v', s', d'', i' \rangle \end{array}}{P \vdash \langle p \text{ delegates } x \ * \ c; S, v, s, d, i \rangle \to \langle v', s', d'', i' \rangle}$$

The remaining three deduction rules specify further delegation by a principal previously authorized by the owner (the head of an existing delegation chain). Note that if a principal only delegates to a sub-chain (of the original chain specified by the owner), only the sub-chain will be added for further delegation.

$$\frac{\begin{array}{c} (c', j) \in d_r(x) \quad c' = c_0 \frown p \frown c \frown c_1 \quad c'(j) = p \\ d' = d \oplus \{x \mapsto (d_w(x), d_r(x) \setminus \{(c', j)\} \cup \{(c', j+1)\})\} \\ P \vdash \langle S, v, s, d', i \rangle \to \langle v', s', d'', i' \rangle \end{array}}{P \vdash \langle p \text{ delegates } x \ * \ c; S, v, s, d, i \rangle \to \langle v', s', d'', i' \rangle}$$

The above deduction rule requires a chain (c') to be present in the set of read delegation chains of x such that p appears at the current index of delegation (is the last actual reader in the chain) in c', and p is immediately followed by c. If such is the case, the current delegation index in c' is incremented by 1, thereby granting the read right to the first principal in c. (Delegating to an empty list shifts an arbitrary delegation chain starting with p one principal further. If this is considered undesired, it can be prevented by adding a premise requiring c to be non-empty.) The other two deduction rules, specifying the delegation of writing and both reading and writing, are almost identical to the one given above and are omitted here.

Example 5 Now we have sufficient ingredients to go somewhat further (than Example 4) with the evaluation of the program in Example 3. To start with, consider a chunk of this program, given below.

```
history : (String, patient) := SQLQuery;
patient delegates history(*,!) phys1;
obs1 : (String, phys1 patient) := Observation;
history = history + obs1;
```

We have already seen that the execution of variable declaration results in the following environment.

$$
\begin{aligned}
\mathit{env_0}=&(\{\mathit{history} \mapsto \mathit{SQLQuery}\}, \{\mathit{history} \mapsto (\{\mathit{patient}\}, \emptyset, \emptyset)\},\\
&\{\mathit{history} \mapsto (\emptyset, \emptyset)\}, \{\mathit{history} \mapsto \{\mathit{patient}\}\})
\end{aligned}
$$

Hence, we can continue with evaluating the rest of the program by using the empty set as PC and env as environment. evaluating the delegation statement results in the following environment, due to the third deduction rule for delegation (note that *patient* is the owner of *history*):

$$
\begin{aligned}
\mathit{env_1} = &(\{\mathit{history} \mapsto \mathit{SQLQuery}\}, \{\mathit{history} \mapsto (\{\mathit{patient}\}, \emptyset, \emptyset)\},\\
&\{\mathit{history} \mapsto (\{(\mathit{phys_1}, 0)\}, \{(\mathit{phys_1}, 0)\})\}, \{\mathit{history} \mapsto \{\mathit{patient}\}\})
\end{aligned}
$$

Hence in $\mathit{env_1}$, $\mathit{current}(d_w(\mathit{history})) = \mathit{current}(d_r(\mathit{history}))$ $= \{\mathit{phys_1}\}$. Evaluating the variable declaration adds obs_1 to the set of declared variables with its initial value, i.e., the following environment:

$$
\begin{aligned}
&\mathit{env_2} = (\{\mathit{history} \mapsto \mathit{SQLQuery}, \mathit{obs1} \mapsto \mathit{Observation}\},\\
&\{\mathit{history} \mapsto (\{\mathit{patient}\}, \emptyset, \emptyset), \mathit{obs1} \mapsto (\{\mathit{phys1}, \mathit{patient}\}, \emptyset, \emptyset)\},\\
&\{\mathit{history} \mapsto (\{(\mathit{phys_1}, 0)\}, \{(\mathit{phys_1}, 0)\})\},\\
&\{\mathit{history} \mapsto \{\mathit{patient}\}, \mathit{obs_1} \mapsto \{\mathit{patient}, \mathit{phys_1}\}\})
\end{aligned}
$$

Considering the assignment statement, it does not hold that $s(\mathit{history}) \sqsubseteq s(\mathit{history} + \mathit{obs}1)$, because only *patient* is authorized by its static label to write to *history*, while *history + obs1* is influenced by $\mathit{phys_1}$. However, it does hold that $\{\mathit{patient}, \mathit{phys_1}\} = \mathit{writers}(s(\mathit{history})) \cup \mathit{current}(d_w(\mathit{history}))$ $\sqsubseteq P \cup i(\mathit{history} + \mathit{obs_1}) = \{\mathit{patient}, \mathit{phys_1}\}$. Hence, the second deduction rule for assignment applies, resulting in the following environment.

$$
\begin{aligned}
&\mathit{env_3} = (\{\mathit{history} \mapsto \mathit{SQLQuery} + \mathit{Observation},\\
&\qquad \mathit{obs1} \mapsto \mathit{Observation}\},\\
&\{\mathit{history} \mapsto (\{\mathit{patient}\}, \emptyset, \emptyset), \mathit{obs1} \mapsto (\{\mathit{phys1}, \mathit{patient}\}, \emptyset, \emptyset)\},\\
&\{\mathit{history} \mapsto (\{(\mathit{phys_1}, 0)\}, \{(\mathit{phys_1}, 0)\})\},\\
&\{\mathit{history} \mapsto \{\mathit{patient}, \mathit{phys_1}\}, \mathit{obs_1} \mapsto \{\mathit{patient}, \mathit{phys_1}\}\})
\end{aligned}
$$

3.8 Revocation statements

The owner of a variable can revoke any delegation (sub-)chain declared before; the grantor of a delegation can revoke only the part of delegation authorized by herself. The grantor cannot revoke the delegation chain defined by the owner, however.

$$
\frac{\begin{aligned}
d' = (d_w, d_r \oplus \{x \mapsto \{((d_r(x)\backslash\\
\{(c', j) \mid (c', j) \in d_r(x) \wedge c' = c_0 \frown c \frown c_1\}) \cup\\
\{(c_0, k) \mid (c_0 \frown c \frown c_1, j) \in d_r(x) \wedge k = \mathit{min}(j, \#c_0)\})\}\})\\
p \in \mathit{owner}(s(x)) \quad P \vdash \langle S, v, s, d', i\rangle \rightarrow \langle v', s', d'', i'\rangle
\end{aligned}}{P \vdash \langle p \text{ revokes } x \ * \ c; S, v, s, d, i\rangle \rightarrow \langle v', s', d'', i'\rangle}
$$

The deduction rule given above, takes out all chains containing the sub-chain c and puts back their prefix to the index right before the sub-chain in case it has gone beyond the prefix. An alternative semantics for revocation could put back $c_0 \frown c_1$, thereby "patching" the remaining chains; this alternative semantics is more in-line with a transitive view of delegation, i.e., if the chain $p_0 \rightarrow p_1 \rightarrow p_2$ is authorized, then the delegation $p_0 \rightarrow p_2$ is also implicitly authorized. We consider this somewhat counter-intuitive as it is quite natural to require a certain delegation to go via a specified principal. (Revoking an empty chain revokes all delegation chains for variables owned by x.)

$$
\frac{\begin{aligned}
d'=(d_w, d_r \oplus \{x \mapsto ((d_r(x)\backslash\\
\{(c', i) \mid (c', i) \in d_r(x) \wedge c' = c_0 \frown p \frown c_1 \frown c \frown c_2 \wedge\\
\#c_0 \frown p \frown c_1 < i\}) \cup\\
\{(c_0 \frown p \frown c_1, \#c_0 \frown p \frown c_1) \mid\\
(c', i) \in d_r(x) \wedge c' = c_0 \frown p \frown c_1 \frown c \frown c_2 \wedge\\
\#c_0 \frown p \frown c_1 < i\})\}\})\\
p \notin \mathit{owner}(s(x)) \quad P \vdash \langle S, v, s, d', i\rangle \rightarrow \langle v', s', d'', i'\rangle
\end{aligned}}{P \vdash \langle p \text{ revokes } x \ * \ c; S, v, s, d, i\rangle \rightarrow \langle v', s', d'', i'\rangle}
$$

The rule given above removes those chains containing p and followed by c such that the current delegation has passed via p to c (and possibly beyond). After removing such chains, their prefix before reaching c is put back in the delegation chain and the index is moved back to the index before the start of c. (Revoking an empty chain revokes all chains that has gone beyond p and returns the delegation to p.)

There are two pairs of deduction rules dedicated to revoking write and both write and read delegation, which are almost identical to the ones given above and hence, are omitted.

3.9 Termination

Finally, an empty program, denoted by ϵ, indicates successful termination, whose semantics is captured by the following deduction rule. $\dfrac{}{P \vdash \langle \epsilon, v, s, d, i\rangle \rightarrow \langle v, s, d, i\rangle}$

4. PROPERTIES

Defining a formal semantics is the first step into the world of formal analysis and hence for the formal analysis to make sense, the definition of formal semantics has to be shown "correct". This is non-trivial, because there is no formal specification, against which the formal semantics can be checked. The only possible specifications are intuitive properties, which are themselves to be formalized. One such intuitive property is the notion of non-interference, which requires that a secure program should produce the same public ("low") state, from any two states that have the same valuation on public variables (which may be arbitrarily different on the valuation of private or "high" ones). (As it is customary in the literature, we assume throughout this section that we only have two logical principals "low" and "high".)

Non-interference is clearly inappropriate for our setting as the valuation of high variables may justifiably influence that of low variables due to a delegation; in other words, high and low only represent the static label of variables and any appropriate notion of security should also cater for the dynamic labels and information flow. There are two alternatives for this: either one resorts to a bisimulation-like definition for non-intereference (e.g., low-bisimulation of [19]), where dynamic labels and information flow are tracked in each operational steps and taken into account in the definition of non-intereference. An alternative approach is to define a notion of non-interference parameterized by a static context of delegation; similar approaches in the literature include the notion indistinguishability relations in [5] and delimited information release in [17]. The latter approach is more semantic-independent and can thus serve as a measure for checking the intuitive properties of the semantics. However, it is much more restrictive as it requires a static delegation context, e.g., a sequence of delegation statements that are executed initially in all runs of the program. To remedy this, we aim to define a compositionality theorem for such a parameterized notion of non-interference, e.g., prove that if two programs are non-interfering, then their sequential and conditional composition is non-interfering as well (we may require some further constraints, matching the parameters of non-interference). Subsequently, by decomposing the program into pieces with static delegation structures, proving their parameterized non-intereference and combining them back again, we can provide a proof of non-interference for complete programs with a dynamic delegation and revocation structure. The details of this approach are yet to be worked out though. Once we have such a notion of security, we should prove the following theorem:

> A program evaluates in our semantics if and only if it is secure.

Another (less involved) property that follows from our intuitive description of delegation, and can be checked for our semantics, is the following:

> A program containing delegation evaluates, only if the delegation is performed by the owner, or by the delegation statements preceding it.

5. CONCLUSIONS

In this paper, we have added the notions of restricted delegation and revocation to a simple imperative programming language and presented their formal semantics. Then, we presented some initial ideas regarding extending the notion of non-interference to cater for delegation and revocation.

Our first research goal is to formalize the ideas presented in Section 4 and to prove them for our proposed semantics. Moreover, we are currently implementing our approach and building up a servers-side scripting language based on our approach (partly, along the lines of [1, 3, 11]). The semantics of [11, 1] provide a semantic alternative to the one presented in this paper, namely, to push the necessary checks for information flow to a concurrent monitor. This approach has the advantage of keeping the programming language and its semantics simple, but requires much communication and synchronization with the monitor, particularly in our case. We are currently investigating this alternative and its practical implications for our ongoing implementation effort.

6. REFERENCES

[1] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proc. of CSF'09*, pp. 43–59. IEEE, 2009.

[2] J. Bacon, K. Moody, and W. Yao. A model of OASIS role-based access control and its support for active security. *ACM TISSEC*, 5(4):492–540, 2002.

[3] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proc. S&P'08*, pp. 387–401. IEEE, 2008.

[4] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15:131–177, 2005.

[5] G. Barthe, P.R. D'Argenio and T. Rezk. Secure Information Flow by Self-Composition. In *Proc. of CSFW'04*, pp. 100-114, IEEE, 2004.

[6] G. Boudol. Secure information flow as a safety property. In *Proc. FAST'08*, volume 5491 of *LNCS*, pp. 20–34. Springer, 2008.

[7] N. Broberg and D. Sands. Paralocks: role-based information flow control and beyond. In *Proc. POPL'2010*, pp. 431-444, ACM, 2010.

[8] S. Chong, A. C. Myers, K. Vikram, and L. Zheng. Jif reference manual, 2006.

[9] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. of S&P'82*, pp. 11–20. IEEE CS, 1982.

[10] G. Kahn. Natural semantics. In *Proc. STACS'87*, volume 247 of *LNCS*, pp. 22–39. Springer, 1987.

[11] G. Le Guernic, A. Banerjee, T. P. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *Proc. of ASIAN'06*, vol. 4435 of *LNCS*, pp. 75–89, Springer, 2008.

[12] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic updating of information-flow policies. In *Proc. of FCS'05*, pp. 7-18, 2005.

[13] H. Mantel and D. Sands. Controlled declassification based on intransitive noninterference. In *Proc. APLAS'04*, volume 3302 of *LNCS*, pp. 129–145. Springer, 2004.

[14] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proc. of ICISC'05*, vol. 3935 of *LNCS*, pp. 156–168. Springer, 2005.

[15] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. POPL'99*, pp. 228–241. ACM, 1999.

[16] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM TOSEM*, 9(4):410–442, 2000.

[17] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. of ISSS'03*, vol. 3233 of *LNCS*, pp. 174–191. Springer, 2003.

[18] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J-SAC*, 21(1):5–19, 2003.

[19] A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-Threaded Programs. In *Proc. of CSFW'00*, pp. 200-214, IEEE, 2004.

[20] J. M. Spivey. *The Z Notation: A Reference Manual, Second Edition.* Prentice Hall, 2001.

[21] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. *ACM TOPLAS*, 30(1):6, 2008.

[22] J. Wainer, A. Kumar, and P. Barthelmess. DW-RBAC: A formal security model of delegation and revocation in workflow systems. *Information System Journal*, 32:365–384, 2007.