Relative Secrecy and Semantics of Declassification*

Peeter Laud
Tartu University
peeter_l@ut.ee

Abstract

This paper introduces the notion of *relatively secure information flow*. A program has relatively secure information flow if its public outputs give no more information about its secret inputs than its *insensitive outputs* do. We give the definition of this property and a program analysis for checking programs for this property. We also show that the effects of declassification of high-security information can be cast as a case of relative secrecy; this allows us mathematically rigorously describe, what kind of information flows declassification introduces and what kind of flows it does not introduce.

1 Introduction

When the inputs and outputs of a program are partitioned to different security classes, we may speak of the problem of *secure information flow*. A program is said to have secure information flow if its low-security outputs provide no help in determining the values of its high-security inputs. Typically, when speaking of programs written in an imperative language, the inputs and outputs of a program are the initial and final values of its variables, respectively. A partition to security classes is defined on the set of variables; this partition defines the high-security inputs and low-security outputs. In this (or in some closely related) formulation, the problem of secure information flow has been well-researched, program analyses have been devised for several possible semantices of the program and for different meanings of "providing no help".

Instead of using a program analysis to certify the program for secure information flow, we may just want to develop the program in a secure way,

*Supported by Estonian Science Foundation, grant #5279

such that insecure information flows just cannot appear. Extensions for programming languages have appeared to faciliate this task, they usually consist of augmenting the type system of the programming language with security annotations. In the simplest case, each variable, expression and statement becomes the type of either "low (security)" or "high", the type system then rules out the assignment of a high expression to a low variable and any assignments to low variables in statements whose execution depends upon the value of some high expression. JFlow [4] is such an extension of the programming language Java that we have particularly in mind.

For "real" programs, such type system may be too restrictive. Although the type of some expression is "high", we may know that its value really does not help finding the values of high inputs. This knowledge may stem from some properties of the distribution of program's inputs and of the operations applied to them that we cannot express in our type system. Alternatively, we may accept the leakage of secret information to the extent that publishing the value of this expression causes; such acceptable leakage may happen when the secret information consists of some database containing personal data, and the aforementioned expression computes some overall statistics of that data. For this reason, the extensions typically provide some kind of *declassify*-expression or -statement. Applying *declassify* to some value results in the same value, but its type is now considered to be low. I.e. the *declassify*-statement is a means to "override" the type system. Unfortunately this means that its introduction has taken away the possibility to prove the soundness of the type system.

In this paper we consider an extension to the problem of secure information flow. Besides the set of low-security outputs we have fixed another subset of program's outputs — the *insensitive* outputs (these two subsets may intersect). We say that a program has *relatively secure information flow* if the low-security outputs give no further information about the high-security inputs than the insensitive outputs do. We give an analysis for certifying the programs for this property. We also show that the security of programs containing *declassify*-statements may be phrased as a problem of relatively secure information flow, this phrasing gives a certain semantic meaning to the *declassify*-statements. This phrasing also helps to explain, exactly how much information is leaked through *declassify*-statements.

After reviewing some related work in Sec. 2, we introduce the simple programming language that we work with in Sec. 3. In Sec. 4 we give the definition of relative secrecy. The next two sections present the program analysis for this property, Sec. 5 introduces the domains that the analysis uses and Sec. 6 gives the transfer functions and also the proof of correctness ("if the analysis says that the program has relatively secure information flow then it really does have it"). Sec. 7 describes the *declassify*-statement and gives a program analysis, similar in precision to the one used for JFlow [4]. Of course, the actual analysis used for JFlow can express much more complex security policies, but if we restrict it to a single principal then it becomes similar to the analysis given in Sec. 7. In Sec. 8 we show how rewrite the *declassify*-statements (i.e. they disappear), such that the program analysis for relative secrecy, applied to the modified program, is at least as precise as the analysis from Sec. 7 on the original program. Finally, Sec. 9 concludes and points out some potential uses of relative secrecy.

2 Related Work

The problem of secure information flow is well-known, its research was started by Denning [1, 2] in the '70s. The basic analysis for secure information flow was recast as a type inference problem by Volpano et al. [9], they were the first to give a correctness proof of the analysis with respect to a security definition that only considers the inputs and outputs of the program, assuming the program itself to be a black box. A recent paper by Sabelfeld and Myers [5] gives a good overview of the research of secure information flow.

A particular instance of relative secrecy has been investigated by Volpano and Smith [8, 7]. They are only considering the problem where a one-way function is given and the image of a certain single high-security value is considered to be low-security. In this case the program may still be secure (if the one-way function really is one-way); they give an analysis for verifying whether programs are secure in this case. We are unaware of any more general approaches to relativizing the notion of secure information flow, however, although the notion of mutual information is well-known in the information theory [6].

When *declassify*-statements or expressions are added to the language then one is interested, how much of the security guarantees can be preserved. We obviously cannot have non-intereference any more, but maybe we can have something weaker. We may also restrict the usage of *declassify*-s if necessary. Zdancewic and Myers [11] have introduced *robust declassification*; a system is robust if an attacker is unable to exploit declassification channels to obtain more information than was intended to be released. Briefly, it means that an active attacker (one that may influence how the program runs) is no more powerful than a passive attacker. Zdancewic [10] gives a type system enforcing robust declassification. The type system requires that *integrity* constraints are added to security types; declassification is allowed only if the execution of the *declassify*-expression depends only on values with high integrity. The type system ensures that the declassified data does not interfere with low-security data, but it does not say anything about its relationship to high-security data. In contrast to their work, we attempt to exactly characterise, how much the declassified data and high-security data interfere.

3 Program Language

We consider a simple imperative programming language (the WHILE-language). The programs P in this language are defined by the following grammar.

$$\begin{array}{rcl} \mathsf{P} & ::= & x := o(x_1, \ldots, x_k) \\ & | & \mathit{skip} \\ & | & \mathsf{P}_1; \mathsf{P}_2 \\ & | & \mathit{if}\ b\ \mathit{then}\ \mathsf{P}_1\ \mathit{else}\ \mathsf{P}_2 \\ & | & \mathit{while}\ b\ \mathit{do}\ \mathsf{P}_1 \end{array}$$

Here $x, x_1, \ldots, x_k, b \in \mathbf{Var}$, $o \in \mathbf{Op}$; $\mathbf{Var}$ is a fixed set of variables and $\mathbf{Op}$ a fixed set of operations (arithmetic, relational, boolean, etc.). Each element of $\mathbf{Op}$ has a fixed arity.

We consider the denotational semantics of that language, the denotational semantics of an imperative programming language maps its initial state to its final state. The state of the program maps the variables to their values, its type is therefore $\mathbf{Var} \to \mathbf{Val}$, where $\mathbf{Val}$ is the set of possible values; we do not specify this set further here. Let $\mathbf{State}$ be the set of all program states. The type of the semantics $[\![\mathsf{P}]\!]$ is $\mathbf{State} \to \mathbf{State}_\perp$, here $\mathbf{State}_\perp = \mathbf{State} \cup \{\perp\}$, the element $\perp$ denotes nontermination.

For each $o \in \mathbf{Op}$ of arity k, let a mapping $[\![o]\!] : \mathbf{Val}^k \to \mathbf{Val}$ be given. The semantics of $x := o(x_1, \ldots, x_k)$, applied to some program state S, only changes the value of the variable x in that state, making it equal to $[\![o]\!](S(x_1), \ldots, S(x_k))$. The semantics of *skip* is the identity function and the semantics of $\mathsf{P}_1; \mathsf{P}_2$ the composition of semantices of P_1 and P_2. The semantics of $\mathit{if}\ b\ \mathit{then}\ \mathsf{P}_1\ \mathit{else}\ \mathsf{P}_2$, applied to S, applies either $[\![\mathsf{P}_1]\!]$ or $[\![\mathsf{P}_2]\!]$ to S, depending on the value of $S(b)$. Finally, the semantics of $\mathit{while}\ b\ \mathit{do}\ \mathsf{P}$ is the least solution to

$$[\![\mathit{while}\ b\ \mathit{do}\ \mathsf{P}]\!] = [\![\mathit{if}\ b\ \mathit{then}\ (\mathsf{P}; \mathit{while}\ b\ \mathit{do}\ \mathsf{P})\ \mathit{else}\ \mathit{skip}]\!]$$

(the order on $\mathbf{State} \to \mathbf{State}_\perp$ is defined pointwise, the order on $\mathbf{State}_\perp$ is defined by considering the elements of **State** incomparable to each other and $\perp$ smaller than any other element).

In the rest of this paper, we ignore the possibility that programs may not terminate. Indeed, the nontermination of programs cannot be observed, we can only observe whether the program run exceeds some time limit. If we want to take into account the possible leakage of information through long running time, we modify the program, making it run parallel to (i.e. interleaved with) a clock, and terminate the program when it has exceeded its time limit. We then analyse the modified program. Hence we assume that the type of $[\![\mathsf{P}]\!]$ is $\mathbf{State} \to \mathbf{State}$.

4 Secrecy Definition

We are not going to partition the set of variables into high- and low-security variables, we will just define the following subsets of the set **Var**:

- $\mathbf{Var}_\mathrm{S}$ — the set of high-security inputs. The initial values of these variables are considered to be secret.

- $\mathbf{Var}_\mathrm{T}$ — the set of other inputs. This means $\mathbf{Var}_\mathrm{T} \cap \mathbf{Var}_\mathrm{S} = \emptyset$. However, this does not mean $\mathbf{Var}_\mathrm{T} \cup \mathbf{Var}_\mathrm{S} = \mathbf{Var}$ — we also allow the program to have some "work" variables. But as $\mathbf{Var}_\mathrm{S}$ and $\mathbf{Var}_\mathrm{T}$ together make up all the inputs of the program, we demand the value of $[\![\mathsf{P}]\!](S)$ to be determined by $S|_{\mathbf{Var}_\mathrm{S} \cup \mathbf{Var}_\mathrm{T}}$.

- $\mathbf{Var}_\mathrm{P}$ — the set of low-security outputs. The final values of these variables are considered to become public.

- $\mathbf{Var}_\mathrm{IS}$ — the set of insensitive outputs. The final values of these variables are considered to contain no information about the secret inputs that we would like to hide.

The usual definition of secure information flow demands that the public outputs do not depend on secret inputs, i.e. the public outputs are determined by other inputs, i.e. there exists a function $f : (\mathbf{Var}_\mathrm{T} \to \mathbf{Val}) \to (\mathbf{Var}_\mathrm{P} \to \mathbf{Val})$, such that $f(S|_{\mathbf{Var}_\mathrm{T}}) = [\![\mathsf{P}]\!](S)|_{\mathbf{Var}_\mathrm{P}}$ for all $S \in \mathbf{State}$. In case of relative secrecy, the public outputs may also contain whatever the insensitive outputs contain, i.e. there must exist a function

$$f : (\mathbf{Var}_\mathrm{T} \to \mathbf{Val}) \times (\mathbf{Var}_\mathrm{IS} \to \mathbf{Val}) \to (\mathbf{Var}_\mathrm{P} \to \mathbf{Val})$$

satisfying

$$f(S|_{\mathbf{Var}_\mathrm{T}}, [\![\mathsf{P}]\!](S)|_{\mathbf{Var}_\mathrm{IS}}) = [\![P]\!](S)|_{\mathbf{Var}_\mathrm{P}}$$

for all $S \in \mathbf{State}$.

For the rest of this paper, let us assume that the program P does not change the values of the variables in $\mathbf{Var}_\mathrm{T}$. Then the last definition is equivalent to the existence of a function f, such that

$$f([\![P]\!](S)|_{\mathbf{Var}_\mathrm{T}}, [\![\mathsf{P}]\!](S)|_{\mathbf{Var}_\mathrm{IS}}) = [\![P]\!](S)|_{\mathbf{Var}_\mathrm{P}} \ . \tag{1}$$

This requirement is somewhat easier to reason about, because it speaks about one program point only (the end of the program) instead of two (the beginning and the end).

For $X \subseteq \mathbf{Var}$ and $S_1, S_2 \in \mathbf{State}$ let $S_1 =_X S_2$ denote $\forall x \in X : S_1(x) = S_2(x)$. The existence of f, such that (1) holds is then equivalent to

$$\begin{aligned}\forall S_1, S_2 \in \mathbf{State} : [\![\mathsf{P}]\!](S_1) =_{\mathbf{Var}_\mathrm{T}} [\![\mathsf{P}]\!](S_2) \wedge [\![\mathsf{P}]\!](S_1) =_{\mathbf{Var}_\mathrm{IS}} [\![\mathsf{P}]\!](S_2) \\ \Rightarrow [\![\mathsf{P}]\!](S_1) =_{\mathbf{Var}_\mathrm{P}} [\![\mathsf{P}]\!](S_2) \ .\end{aligned} \tag{2}$$

5 Abstract Domain

We abstract sets of program states (in the end, we are interested whether the set of all final program states has a certain property, given by (2)).

Let a set of program states $\mathcal{S} \subseteq \mathbf{State}$ be given. For all $X, Y, Z \subseteq \mathbf{Var}$ we are interested whether

$$\forall S_1, S_2 \in \mathcal{S} : S_1 =_X S_2 \wedge S_1 =_Y S_2 \Rightarrow S_1 =_Z S_2 \tag{3}$$

holds. Initiating $\mathcal{S}$ with the set of all final program states, X with $\mathbf{Var}_\mathrm{T}$, Y with $\mathbf{Var}_\mathrm{IS}$ and Z with $\mathbf{Var}_\mathrm{P}$ gives us (2).

For knowing whether (3) holds for each triple (X, Y, Z), it is obviously enough to know for each $X \subseteq \mathbf{Var}$ and $z \in \mathbf{Var}$, whether

$$\forall S_1, S_2 \in \mathcal{S} : S_1 =_X S_2 \Rightarrow S_1(z) = S_2(z) \tag{4}$$

holds or not. We take $\mathbf{AD}(\mathbf{Var}) = \mathcal{P}(\mathcal{P}(\mathbf{Var}) \times \mathbf{Var})$ as our abstract domain. We have the abstraction function $\alpha : \mathcal{P}(\mathbf{State}) \to \mathbf{AD}(\mathbf{Var})$ that maps each $\mathcal{S} \subseteq \mathbf{State}$ to the set of all pairs (X, z) satisfying (4). We say that some $A \in \mathbf{AD}(\mathbf{Var})$ *abstracts* some $\mathcal{S} \subseteq \mathbf{State}$ if $A \subseteq \alpha(\mathcal{S})$. In the next section we define the program analysis $\mathcal{A}(\mathsf{P})$ of the program P. It is a function whose domain and range are both $\mathbf{AD}(\mathbf{Var})$. It has the property that if A abstract the set of possible initial states of P then $\mathcal{A}(\mathsf{P})(A)$ abstracts the set of possible final states.

The property (4) means that the values of variables in the set X determine the value of the variable z; or that the value of z can be found (not necessarily efficiently or even algorithmically) from the values of variables in X. Let $\mathcal{S} \subseteq \mathbf{State}$ and $A = \alpha(\mathcal{S})$. It is obvious that A has the following properties:

(i) $(\{z\}, z) \in A$ for all $z \in \mathbf{Var}$. Indeed, the value of z can always be found from the value of z.

(ii) If $(X, z) \in A$ and $Y \subseteq \mathbf{Var}$, then $(X \cup Y, z) \in A$. The value of z can be found from the values of $X \cup Y$ in the same way as from the values of X, by first discarding the values of variables in $Y \backslash X$.

(iii) If $(X \cup \{y\}, z) \in A$ and $(X, y) \in A$, then also $(X, z) \in A$. The value of z can be found from the values of X by first finding the value of y from the values of X and then finding the value of z from the values of $X \cup \{y\}$.

Let $A \in \mathbf{AD}(\mathbf{Var})$. We say that A is *closed* if it satisfies the three properties above. The *closure* $\overline{A} \in \mathbf{AD}(\mathbf{Var})$ of A is the smallest closed set containing A. Each $A \in \mathbf{AD}(\mathbf{Var})$ has the closure because the intersection of any number of closed sets is obviously closed and $\mathcal{P}(\mathbf{Var}) \times \mathbf{Var}$ — the largest element of $\mathbf{AD}(\mathbf{Var})$ — is closed as well. It should also be clear how to find the closure of some A — just keep adding to it the elements that these three properties require it to contain.

6 Program Analysis for Relative Secrecy

The analysis $\mathcal{A}(\mathsf{P})$ of a program P has the type $\mathbf{AD}(\mathbf{Var}) \to \mathbf{AD}(\mathbf{Var})$. It is defined inductively over program structure (we could also define it with help of the control flow graph). In this section we give its definition and, in parallel, also its proof of correctness (the abstraction of the set of possible input states is mapped to an abstraction of possible output states). The correctness of the analysis allows its use for checking the programs for relatively secure information flow in the following way. Let P, $\mathbf{Var}_{\mathrm{T}}$, $\mathbf{Var}_{\mathrm{IS}}$ and $\mathbf{Var}_{\mathrm{P}}$ be given.

- Apply $\mathcal{A}(\mathsf{P})$ to the abstraction $A\circ$ of the set of possible input states to P, yielding $A_\bullet$. If nothing is known about the set of possible input states then assume that all elements of **State** may belong to it. In this case, $A_\circ = \overline{\emptyset}$.
- Verify, whether $(\mathbf{Var}_{\mathrm{T}} \cup \mathbf{Var}_{\mathrm{IS}}, z) \in A_\bullet$ for all $z \in \mathbf{Var}_{\mathrm{P}}$. If yes, then output "program has relatively secure information flow", else output "don't know".

Consider the outermost constructor of P.

1st case. P is $x := o(x_1, \ldots, x_k)$; we assume w.l.o.g. that x is different from $x_1, \ldots, x_k$ (if this is not so, then introduce a temporary variable to hold the value of $o(x_1, \ldots, x_k)$ before assigning it to x). The analysis $\mathcal{A}(\mathsf{P})$, applied to $A = A_\circ$, works as follows:

1. kill x, i.e. remove from A all pairs (X, z), where $x \in X$ or $x = z$;
2. add $(\{x_1, \ldots, x_k\}, x)$ to A;
3. take $A_\bullet = \overline{A}$.

The analysis is correct — before the 3rd step, the pairs $(X, z) \in A$ have two possible forms:

- $x \notin X \cup \{z\}$. Because of the shape of $\mathcal{A}(\mathsf{P})$, also $(X, z) \in A_\circ$. As $A_\circ$ abstracted the set of possible input states, the value of z could be found from the values of X before running the program P. The program P did not change the values of variables in $X \cup \{z\}$, therefore the value of z can be found from the values of X after running the program.
- $X = \{x_1, \ldots, x_k\}$ and $z = x$. The value of z can be found from the values of X after running the program by applying o.

We have shown that the first two steps of $\mathcal{A}(\mathsf{P})$ constitute a correct program analysis. Forming the closure obviously does not make an otherwise correct analysis incorrect.

If the operation o is somehow invertible, then we may add more pairs during the second step of computing $A_\bullet$. For example, if the program P was $x := y + z$, then we may also add pairs $(\{x, y\}, z)$ and $(\{x, z\}, y)$ during this step. In general, if some x_i is computable from some $X \subseteq \{x, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k\}$ after the operation $x := o(x_1, \ldots, x_k)$, then we add the pair (X, x_i) to A during the second step of computing $A_\bullet$. The

correctness of adding of these pairs follows similarly to the correctness of adding $(\{x_1,\ldots,x_k\},x)$.
2nd case. P is *skip*. Then $\mathcal{A}(\mathsf{P})$ is the identity function over $\mathbf{AD}(\mathbf{Var})$.
3rd case. P is $\mathsf{P}_1;\mathsf{P}_2$. Then $\mathcal{A}(\mathsf{P})=\mathcal{A}(\mathsf{P}_2)\circ\mathcal{A}(\mathsf{P}_1)$. The correctness of these two cases should be obvious.
4th case. P is *if* b *then* P_1 *else* P_2. Let $\mathbf{Var}_{\mathrm{asgn}}\subseteq\mathbf{Var}$ be the set of variables that stand at the left hand side of some assignment in P_1 or P_2. Let N be a variable that is not a member of $\mathbf{Var}$ and let $\mathbf{Var}'=\mathbf{Var}\cup\{N\}$. The analysis $\mathcal{A}(\mathsf{P})$, applied to $A_\circ$, first computes $A_1=\mathcal{A}(N:=b;\mathsf{P}_1)(A_\circ)$ and $A_2=\mathcal{A}(N:=b;\mathsf{P}_2)(A_\circ)$; they are computed over the domain $\mathbf{AD}(\mathbf{Var}')$. Let $A=A_1\cap A_2$. A pair $(X,z)\in\mathbf{AD}(\mathbf{Var})$ is taken to the analysis result $A_\bullet$ if the following conditions hold:

- $(X,z)\in A$;
- if $\mathbf{Var}_{\mathrm{asgn}}\cap\big(X\cup\{z\}\big)\neq\emptyset$ then $(X\backslash\mathbf{Var}_{\mathrm{asgn}},N)\in A$.

Indeed, if $(X,z)\in A$, then the value of z can be found from the values of X at the ends of both branches. Of course, the exact way of determining the value of z may be different in both branches. To be able to find the value of z at the end of the program P (i.e. after the confluence of branches P_1 and P_2), either the way of finding the value of z from the values of X must be independent of the chosen branch, or we must be able to determine (from the values of X) which branch was taken. If neither z nor any variable in X was changed in neither branch, then the way of finding the value of z from the values of X is the same in both branches (and also the same as in the beginning of the program P). Otherwise we must determine the taken branch (from the values of X), i.e. we must determine the value of N. If we have $(X,N)\in A$, then the value of N can be found from the values of X at the end of both branches, but we have the same problem again — the way of finding N may be different in both branches and we do not know, which one to use. We solve this by restricting the procedure of finding N to only the values of these variables in X that have not been changed in the branches. Then the same way of finding N will work for both branches.

Remark. The condition $(X\backslash\mathbf{Var}_{\mathrm{asgn}},N)\in A$ is equivalent to the condition $(X\backslash\mathbf{Var}_{\mathrm{asgn}},N)\in A'$, where $A'=\mathcal{A}(N:=b)(A_\circ)$ because neither N nor the variables in $X\backslash\mathbf{Var}_{\mathrm{asgn}}$ are changed in P_1 or P_2. In some cases, this condition may be weakened. Namely, let $d\in\mathbf{Var}_{\mathrm{asgn}}$ be such variable that its value before running the program P can be determined from its value after running P (i.e. this value is taken after the confluence of the branches P_1 and P_2). For example, such determination is possible if both branches change the value of d in the same way (and this change is invertible). Such determination is also possible if the value of d is a list that is only appended to in both branches and the first appended value in either branch is somehow marked.

If $d \in \mathbf{Var}_{\mathrm{asgn}}$ is a variable with the abovementioned property, then we may replace the condition

- if $\mathbf{Var}_{\mathrm{asgn}} \cap \big(X \cup \{z\}\big) \neq \emptyset$ then $(X \backslash \mathbf{Var}_{\mathrm{asgn}}, N) \in A$

with the condition

- if $\mathbf{Var}_{\mathrm{asgn}} \cap \big((X \backslash \{d\}) \cup \{z\}\big) \neq \emptyset$ then $(X \backslash (\mathbf{Var}_{\mathrm{asgn}} \backslash \{d\}), N) \in A'$.

5th case. P is *while* b *do* P_1. Let $\mathbf{Var}_{\mathrm{asgn}} \subseteq \mathbf{Var}$ be defined as in the previous case. Let N and $\mathbf{Var}'$ also have the same meaning as in the previous case. Let $A' = \mathcal{A}(N := b)(A_\circ)$. The analysis $\mathcal{A}(\mathsf{P})$, applied to $A_\circ$, computes $A \in \mathbf{AD}(\mathbf{Var}')$ as the largest set satisfying

$$A = A' \cap \mathcal{A}(\mathsf{P}_1; N := b)(A) \ . \tag{5}$$

Such A can be found iteratively, by initialising A with A' and then repeatedly (as long as there are changes) applying (5), interpreting the equals-sign as the assignment operation. The finiteness of the abstract domain $\mathbf{AD}(\mathbf{Var}')$ guarantees the termination of this approach.

A pair $(X, z) \in \mathbf{AD}(\mathbf{Var})$ is taken to the analysis result $A_\bullet$ if the following conditions hold:

- $(X, z) \in A$;
- if $\mathbf{Var}_{\mathrm{asgn}} \cap \big(X \cup \{z\}\big) \neq \emptyset$ then $(X \backslash \mathbf{Var}_{\mathrm{asgn}}, N) \in A$.

The condition $(X, z) \in A$ ensures the possibility of finding the value of z from the values of X after each iteration. Again, the way of finding this value may be different after each iteration. If neither z nor the variables in X have been changed by the program, then the way of finding the value of z from the values of X obviously does not depend on the number of iterations. Otherwise the condition $(X \backslash \mathbf{Var}_{\mathrm{asgn}}, N) \in A$ ensures that we can find the value of N after each iteration from the values of X at the end of the program P (we are only using that part of X that is unaffected by P). We can now find from the values of X at the end of P the smallest i, such that the value of N was false after the i-th iteration; i is the number of iterations made by P. Now we know, which way of finding the value of z from the values of X to use.

Remark. A weakening similar to the previous case is possible if $d \in \mathbf{Var}_{\mathrm{asgn}}$ is such a variable that from its value at the end of the loop we can determine its value at the beginning of the loop, as well as its value after each iteration. The condition

- if $\mathbf{Var}_{\mathrm{asgn}} \cap \big(X \cup \{z\}\big) \neq \emptyset$ then $(X \backslash \mathbf{Var}_{\mathrm{asgn}}, N) \in A$

may then be replaced by

- if $\mathbf{Var}_{\mathrm{asgn}} \cap \big((X \backslash \{d\}) \cup \{z\}\big) \neq \emptyset$ then $(X \backslash (\mathbf{Var}_{\mathrm{asgn}} \backslash \{d\}), N) \in A$.

Indeed, from the values of $X \backslash (\mathbf{Var}_{\mathrm{asgn}} \backslash \{d\})$ after the loop we can still determine the value of N after each iteration of the loop.

This concludes the presentation of the program analysis $\mathcal{A}$. The analysis should be efficiently implementable; keeping track of the sets of variables will not be trivial but should be doable with help of binary decision diagrams, in the vein of [3, Chapter 5].

7 The *declassify*-statement

The statement *declassify*(x), where $x \in \mathbf{Var}$, is introduced to the program language to inform the analysis for secure information flow that the value of the variable x does not depend on the sensitive inputs, even if the program analysis is unable to deduce this fact itself. The semantics $[\![\mathit{declassify}(x)]\!]$ is equal to $[\![\mathit{skip}]\!]$, but the analysis of these two statements is different.

Let us describe a program analysis $\mathcal{B}$ for secure information flow; the programming language extensions for ensuring secure information flow function similarly to this analysis (they have the same precision). The domain of the analysis is $\mathcal{P}(\mathbf{Var})$; if $B \subseteq \mathbf{Var}$ and the program P does not contain *declassify*-statements, then the analysis guarantees (i.e. it is provable that) that the final values of the variables in $\mathcal{B}(\mathsf{P})(B)$ depend only on the initial values of the variables in B. The program P is deemed secure if $\mathbf{Var}_{\mathrm{P}} \subseteq \mathcal{B}(\mathsf{P})(\mathbf{Var}_{\mathrm{T}})$. With declassification, the criterion for deeming the program secure remains the same, but we lose the soundness proof.

The analysis $\mathcal{B}$ is defined inductively over the program structure. We have

$$\begin{aligned} \mathcal{B}(x := o(x_1,\ldots,x_k))(B) &:= \begin{cases} B \cup \{x\}, & \text{if } x_1,\ldots,x_k \in B \\ B \backslash \{x\}, & \text{otherwise} \end{cases} \\ \mathcal{B}(\mathit{declassify}(x))(B) &:= B \cup \{x\} \ . \end{aligned}$$

$\mathcal{B}(\mathit{skip})$ is again the identity function (over $\mathcal{P}(\mathbf{Var})$) and $\mathcal{B}(\mathsf{P}_1;\mathsf{P}_2)$ is the composition of $\mathcal{B}(\mathsf{P}_1)$ and $\mathcal{B}(\mathsf{P}_2)$.

The analysis $\mathcal{B}(\mathit{if}\ \ b\ \ \mathit{then}\ \mathsf{P}_1\ \ \mathit{else}\ \mathsf{P}_2)$, applied to $B_\circ$, first constructs $B_1 = \mathcal{B}(N := b;\mathsf{P}_1)(B_\circ)$ and $B_2 = \mathcal{B}(N := b;\mathsf{P}_2)(B_\circ)$. Here $B_1, B_2 \subseteq \mathbf{Var}'$, the symbols N and $\mathbf{Var}'$ have the same meaning as in the definition of the analysis $\mathcal{A}$. Let $B = B_1 \cap B_2$; a variable $x \in \mathbf{Var}$ belongs to the analysis result $B_\bullet$ if $x \in B$ and if $x \in \mathbf{Var}_{\mathrm{asgn}}$ then also $N \in B$.

The analysis $\mathcal{B}(\mathit{while}\ \ b\ \ \mathit{do}\ \ \mathsf{P}_1)$, applied to $B_\circ$ first constructs $B' = \mathcal{B}(N := b)(B_\circ)$ and then computes $B \subseteq \mathbf{Var}'$ as the largest set satisfying

$$B = B' \cap \mathcal{B}(\mathsf{P}_1; N := b)(B) \ .$$

A variable $x \in \mathbf{Var}$ belongs to the analysis result $B_\bullet$ if $x \in B$ and if $x \in \mathbf{Var}_{\mathrm{asgn}}$ then also $N \in B$.

8 Declassification as Relative Secrecy

Let a program P, containing *declassify*-statements be given. In this section we introduce a program transformation $\mathcal{T}(\cdot)$, such that if the analysis $\mathcal{B}$ deems P secure then the analysis $\mathcal{A}$ deems $\mathcal{T}(\mathsf{P})$ secure. There will be no *declassify*-statements in the transformed program. Additionally, we have to define the set $\mathbf{Var}_{\mathrm{IS}}$ and the connection between $\mathcal{P}(\mathbf{Var})$ and $\mathbf{AD}(\mathbf{Var})$.

The connection between $\mathcal{P}(\mathbf{Var})$ and $\mathbf{AD}(\mathbf{Var})$ should be quite obvious. By the analysis $\mathcal{B}$, we required each $x \in \mathbf{Var}_{\mathrm{P}}$ to belong to the final analysis information $B_\bullet$. By the analysis $\mathcal{A}$, we required for each $x \in \mathbf{Var}_{\mathrm{P}}$ that the pair $(\mathbf{Var}_{\mathrm{T}} \cup \mathbf{Var}_{\mathrm{IS}}, x)$ belongs to the final analysis information $A_\bullet$. For $B \subseteq \mathbf{Var}$ we define

$$\xi(B) = \overline{\left\{(\mathbf{Var}_{\mathrm{T}} \cup \mathbf{Var}_{\mathrm{IS}}, x) \,|\, x \in B\right\}} \tag{6}$$

(recall that $\bar{\cdot}$ denoted the closure of an element of $\mathbf{AD}(\mathbf{Var})$). The program transformation $\mathcal{T}(\cdot)$ has the following property:

$$\xi\big(\mathcal{B}(\mathsf{P})(B_\circ)\big) \subseteq \mathcal{A}(\mathcal{T}(\mathsf{P}))(\xi(B_\circ)) \tag{7}$$

for all $B_\circ \subseteq \mathbf{Var}$. Taking here $B_\circ = \mathbf{Var}_{\mathrm{T}}$ gives us the statement "if $\mathcal{B}$ deems P secure then $\mathcal{A}$ deems $\mathcal{T}(\mathsf{P})$ secure". Note also that $\xi(\mathbf{Var}_{\mathrm{T}})$ is a subset of every closed element of $\mathbf{AD}(\mathbf{Var})$, including the abstraction of the set of possible input states of P.

We have already mentioned the set $\mathbf{Var}_{\mathrm{IS}}$, but we have not told yet, which variables belong to that set. We will explain it now, as we start to describe the program transformation. Let d be a variable that is not used by the program P, we let $\mathbf{Var}$ contain all the variables used by P, as well as d. Let $\mathbf{Var}_{\mathrm{IS}} = \{d\}$. The value of d will be a list where we basically collect the values declassified during the program run. In the following, we describe the transformation $\mathcal{T}(\cdot)$. In parallel to its description, we also show that (7) holds for all $B_\circ \subseteq \mathbf{Var}$ (in showing that, we make use of induction over the structure of P). In other words, for all $B_\circ \subseteq \mathbf{Var}$ and $z \in \mathbf{Var}$,

$$z \in B_\bullet \Longrightarrow (\mathbf{Var}_{\mathrm{T}} \cup \{d\}, z) \in A_\bullet \tag{8}$$

where $B_\bullet = \mathcal{B}(\mathsf{P})(B_\circ)$ and $A_\bullet = \mathcal{A}(\mathcal{T}(\mathsf{P}))(\xi(B_\circ))$.

If P does not contain *declassify*-statements, then $\mathcal{T}(\mathsf{P}) = \mathsf{P}$. We prove (8) by induction over the structure of P (this proof also shows that $\mathcal{A}$ is at least as precise as the "usual" analyses for secure information flow).

If P is *skip*, then the left hand side of the implication (8) means that $z \in B_\circ$. To show the right hand side, it suffices to show that $(\mathbf{Var}_{\mathrm{T}} \cup \{d\}, z) \in \xi(B_\circ)$. But this is clearly the case by (6).

If P is $x := o(x_1, \ldots, x_k)$, then consider two cases. First case — $z \neq x$. Then $z \in B_\circ$ and $(\mathbf{Var}_{\mathrm{T}} \cup \{d\}, z) \in \xi(B_\circ)$. No variable in this pair is changed by the program P, therefore this pair also belongs to the final analysis information $A_\bullet$.

Second case — $z = x$. Then $x_1, \ldots, x_k \in B_\circ$ (by the definition of $\mathcal{B}$) and

$$(\mathbf{Var}_{\mathrm{T}} \cup \{d\}, x_1) \in \xi(B_\circ) \qquad (\mathbf{Var}_{\mathrm{T}} \cup \{d\}, x_1) \in A_\bullet \tag{9}$$

$$(\mathbf{Var}_{\mathrm{T}} \cup \{d\}, x_2) \in \xi(B_\circ) \qquad (\mathbf{Var}_{\mathrm{T}} \cup \{d\}, x_2) \in A_\bullet \tag{10}$$

$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$

$$(\mathbf{Var}_{\mathrm{T}} \cup \{d\}, x_k) \in \xi(B_\circ) \qquad (\mathbf{Var}_{\mathrm{T}} \cup \{d\}, x_k) \in A_\bullet; \tag{11}$$

the right column follows from the left and from the fact that no variables above are changed by the program (recall that the variables in $\mathbf{Var}_{\mathrm{T}}$ are not assigned to and x is different from $x_1, \ldots, x_k$). The definition of the analysis $\mathcal{A}$ gives $(\{x_1, \ldots, x_k\}, x) \in A_\bullet$. Applying the closure property (ii) gives $(\mathbf{Var}_{\mathrm{T}} \cup \{d, x_1, \ldots, x_k\}, x) \in A_\bullet$. Applying to it and (9) the closure property (iii) gives $(\mathbf{Var}_{\mathrm{T}} \cup \{d, x_2, \ldots, x_k\}, x) \in A_\bullet$. Applying to it and (10) the closure property (iii) gives $(\mathbf{Var}_{\mathrm{T}} \cup \{d, x_3, \ldots, x_k\}, x) \in A_\bullet$, etc., finally, with the help of (11) we get $(\mathbf{Var}_{\mathrm{T}} \cup \{d\}, x) \in A_\bullet$.

If P is $\mathsf{P}_1; \mathsf{P}_2$ then let $B_\circ \subseteq \mathbf{Var}$. We have

$$\xi(\mathcal{B}(\mathsf{P}_1;\mathsf{P}_2)(B_\circ)) = \xi(\mathcal{B}(\mathsf{P}_2)(\mathcal{B}(\mathsf{P}_1)(B_\circ))) \subseteq \mathcal{A}(\mathsf{P}_2)(\xi(\mathcal{B}(\mathsf{P}_1)(B_\circ))) \subseteq \mathcal{A}(\mathsf{P}_2)(\mathcal{A}(\mathsf{P}_1)(\xi(B_\circ))) = \mathcal{A}(\mathsf{P}_1;\mathsf{P}_2)(\xi(B_\circ)),$$

i.e. (7) holds.

If P is *if* b *then* P_1 *else* P_2, then let $B_\circ \subseteq \mathbf{Var}$. According to the definition of $\mathcal{A}$, we have to compute $A_1 = \mathcal{A}(N := b; \mathsf{P}_1)(\xi(B_\circ))$ and $A_2 = \mathcal{A}(N := b; \mathsf{P}_2)(\xi(B_\circ))$. According to the definition of $\mathcal{B}$, we have to compute $B_1 = \mathcal{B}(N := b; \mathsf{P}_1)(B_\circ)$ and $B_1 = \mathcal{B}(N := b; \mathsf{P}_2)(B_\circ)$. By the induction assumption, $\xi(B_1) \subseteq A_1$ and $\xi(B_2) \subseteq A_2$. Next we compute $B = B_1 \cap B_2$ and $A = A_1 \cap A_2$, we have $\xi(B) \subseteq A$. If $z \in B$ then $(\mathbf{Var}_\mathrm{T} \cup \{d\}, z) \in A$. If any of the variables in $\mathbf{Var}_\mathrm{T} \cup \{d, z\}$ belongs to $\mathbf{Var}_\mathrm{asgn}$ then $z \in \mathbf{Var}_\mathrm{asgn}$ because the other variables from that set are not assigned to. We have $(\mathbf{Var}_\mathrm{T} \cup \{d\}) \backslash \mathbf{Var}_\mathrm{asgn} = \mathbf{Var}_\mathrm{T} \cup \{d\}$. If $N \in B$ then $(\mathbf{Var}_\mathrm{T} \cup \{d\}, N) \in A$. Hence if $z \in B_\bullet$ then $(\mathbf{Var}_\mathrm{T} \cup \{d\}, z) \in A_\bullet$.

If P is *while* b *do* P_1, then let $B_\circ \subseteq \mathbf{Var}$. Let $A \in \mathbf{AD}(\mathbf{Var}')$ and $B \subseteq \mathbf{Var}'$ (where $\mathbf{Var}' = \mathbf{Var} \cup \{N\}$) be defined as in the definitions of the analyses $\mathcal{A}$ and $\mathcal{B}$ for *while*-statements. We again have $\xi(B) \subseteq A$ by induction assumption and by the finiteness of the abstract domains. Simliarly to the *if*-statements we get that if $z \in B$ then $(\mathbf{Var}_\mathrm{T} \cup \{d\}, z) \in A$, that $(\mathbf{Var}_\mathrm{T} \cup \{d\}) \backslash \mathbf{Var}_\mathrm{asgn} = \mathbf{Var}_\mathrm{T} \cup \{d\}$ and finally if $z \in B_\bullet$ then $(\mathbf{Var}_\mathrm{T} \cup \{d\}, z) \in A_\bullet$.

Let us continue with the description of $\mathcal{T}$. $\mathcal{T}(\mathit{declassify}(x))$ is equal to $\mathit{tmp} := d; d := (\mathit{tmp}, x)$. Here the temporary variable tmp has been introduced only because we stated earlier that the variable at the left hand side of an assignment may not occur in its right hand side. The implication (8) holds: let $z \in \mathbf{Var}$, consider two cases — either $z = x$ or not. If $z \neq x$ then $z \in B_\bullet$ implies $z \in B_\circ$ implies $(\mathbf{Var}_\mathrm{T} \cup \{d\}, z) \in \xi(B_\circ)$. Let $A' = \mathcal{A}(\mathit{tmp} := d)(\xi(B_\circ))$, then $(\mathbf{Var}_\mathrm{T} \cup \{\mathit{tmp}\}, z) \in A'$. When we analyse the statement $d := (\mathit{tmp}, x)$, we add the pair $(\{\mathit{tmp}, x\}, d)$ to A' to obtain $A_\bullet$. But we also add $(\{d\}, \mathit{tmp})$, as well as $(\{d\}, x)$, because the values of tmp and x can both be found from the value of d after this assignment. From $(\mathbf{Var}_\mathrm{T} \cup \{\mathit{tmp}\}, z) \in A_\bullet$ and $(\{d\}, \mathit{tmp}) \in A_\bullet$, as well as the closedness of $A_\bullet$ follows $(\mathbf{Var}_\mathrm{T} \cup \{d\}, z) \in A_\bullet$.

If $z = x$ then $(\{d\}, x) \in A_\bullet$. From the closedness of $A_\bullet$ follows $(\mathbf{Var}_\mathrm{T} \cup \{d\}, x) \in A_\bullet$.

$\mathcal{T}(\mathsf{P}_1; \mathsf{P}_2)$ is equal to $\mathcal{T}(\mathsf{P}_1); \mathcal{T}(\mathsf{P}_2)$. The inclusion (7) holds:

$$\xi(\mathcal{B}(\mathsf{P}_1; \mathsf{P}_2)(B_\circ)) = \xi(\mathcal{B}(\mathsf{P}_2)(\mathcal{B}(\mathsf{P}_1)(B_\circ))) \subseteq \mathcal{A}(\mathcal{T}(\mathsf{P}_2))(\xi(\mathcal{B}(\mathsf{P}_1)(B_\circ))) \subseteq \\ \mathcal{A}(\mathcal{T}(\mathsf{P}_2))(\mathcal{A}(\mathcal{T}(\mathsf{P}_1))(\xi(B_\circ))) = \mathcal{A}(\mathcal{T}(\mathsf{P}_1); \mathcal{T}(\mathsf{P}_2))(\xi(B_\circ)),$$

For defining $\mathcal{T}$ for *if*- and *while*-statements, we have to assume that these statements are *labelled*. A set of labels **Lab** is introduced and each *if*- and *while*-statement will have one element of **Lab** associated with it; different statements will get different labels. $\mathcal{T}(if^l\ b\ then\ \mathsf{P}_1\ else\ \mathsf{P}_2)$ is equal to the following program:

$$tmp := d; d := (tmp, \star_l); if\ \ b\ then\ \mathcal{T}(\mathsf{P}_1)\ else\ \mathcal{T}(\mathsf{P}_2)\ . \qquad (12)$$

Here l is the label of the *if*-statement and $\star_l \in \mathbf{Val}$ is a value that is not used anywhere else. We have "marked" the beginning of the *if*-statement, this is necessary to make (7) hold. With the mark $\star_l$ in the value of d, we can find the value of d before the *if*-statement from its value after that statement — just delete from the end of the list that is the value of d all elements up to (the last) $\star_l$. When we apply the analysis $\mathcal{A}$ to the program (12), the remark after the definition of $\mathcal{A}$ for *if*-statements is applicable.

Let $z \in \mathbf{Var}$ and $B_\circ \subseteq \mathbf{Var}$, we show that (8) holds. Let $A'' = \mathcal{A}(tmp := d; d := (tmp, \star_l))(\xi(B_\circ))$, then $\xi(B_\circ) \subseteq A''$ because from the value of d after these two assignments we can find the value of d before them and the value of no other variable has changed. To compute $A_\bullet$, we have to compute $A_1 = \mathcal{A}(N := b; \mathcal{T}(\mathsf{P}_1))(A'')$ and $A_2 = \mathcal{A}(N := b; \mathcal{T}(\mathsf{P}_2))(A'')$. To compute $B_\bullet$, we have to compute $B_1 = \mathcal{B}(N := b; \mathsf{P}_1)(B_\circ)$ and $B_2 = \mathcal{B}(N := b; \mathsf{P}_2)(B_\circ)$. By induction assumption, $\xi(B_1) \subseteq A_1$ and $\xi(B_2) \subseteq A_2$. If $z \in B_1 \cap B_2$ then $(\mathbf{Var}_\mathrm{T} \cup \{d\}, z) \in A_1 \cap A_2$, hence if $z \notin \mathbf{Var}_\mathrm{asgn}$ then $z \in B_\bullet$ immediately implies $z \in A_\bullet$. If $z \in \mathbf{Var}_\mathrm{asgn}$ and $z \in B_\bullet$ then $N \in B_1 \cap B_2$ which implies $b \in B_\circ$ and $(\mathbf{Var}_\mathrm{T} \cup \{d\}, b) \in A''$. Denoting $A' = \mathcal{A}(N := b)(A'')$ gives us $(\mathbf{Var}_\mathrm{T} \cup \{d\}, N) \in A'$ that is sufficient for $(\mathbf{Var}_\mathrm{T} \cup \{d\}, z) \in A_\bullet$. Indeed, we have $(\mathbf{Var}_\mathrm{T} \cup \{d\}) \backslash (\mathbf{Var}_\mathrm{asgn} \backslash \{d\}) = \mathbf{Var}_\mathrm{T} \cup \{d\}$ because that variables in $\mathbf{Var}_\mathrm{T}$ are not assigned to.

$\mathcal{T}(while^l\ b\ do\ \mathsf{P})$ is equal to the following program:

$$tmp := d; d := (tmp, \star_l^s); while\ \ b\ \ do\ \mathcal{T}(\mathsf{P}); tmp := d; d := (tmp, \star_l^i) \qquad (13)$$

(the scope of *do* is until the end of the line). Here $\star_l^s, \star_l^i \in \mathbf{Val}$ are two values that are not used anywhere else. With $\star_l^s$ and $\star_l^i$ marking the start of each iteration of the loop in the list that is the value of d, the remark after the definition of $\mathcal{A}$ for *while*-statements is again applicable. Indeed, from the value of d after the loop we can find its value at the beginning of the loop, as well as after each iteration.

Let $z \in \mathbf{Var}$ and $B_\circ \subseteq \mathbf{Var}$, we show that (8) holds. Let $A'' = \mathcal{A}(tmp := d; d := (tmp, \star_l^s))(\xi(B_\circ))$, then $\xi(B_\circ) \subseteq A''$ because from the value of d after these two assignments we can find the value of d before them and the value of no other variable has changed. For the analysis $\mathcal{A}$, we now have to compute $A' = \mathcal{A}(N := b)(A'')$ and for the analysis $\mathcal{B}$ we compute

$B' = \mathcal{B}(N := b)(B_\circ)$. By the induction assumption, $\xi(B') \subseteq A'$. Also, for all $\bar{A} \in \mathbf{AD}(\mathbf{Var}')$ and $\bar{B} \subseteq \mathbf{Var}'$:

$$\xi(\bar{B}) \subseteq \bar{A} \Longrightarrow \xi(\mathcal{B}(\mathsf{P}; N := b)(\bar{B})) \subseteq \mathcal{A}(\mathcal{T}(\mathsf{P}); N := b)(\bar{A}) \tag{14}$$

by the induction assumption, and

$$\xi(\bar{B}) \subseteq \bar{A} \Longrightarrow \xi(\bar{B}) \subseteq \mathcal{A}(\mathit{tmp} := d; d := (\mathit{tmp}, \star_l^i))(\bar{A}) \tag{15}$$

for the same reason as $\xi(B_\circ) \subseteq A''$. We also have

$$\mathcal{A}(N := b; \mathit{tmp} := d; d := (\mathit{tmp}, \star_l^i)) = \mathcal{A}(\mathit{tmp} := d; d := (\mathit{tmp}, \star_l^i); N := b)$$

by the properties of $\mathcal{A}$ (if the variables in two assignments are different then the value of the analysis does not depend on the order that they are performed). Therefore

$$\xi(\mathcal{B}(\mathsf{P}; N := b)(\hat{B})) \subseteq \mathcal{A}(\mathcal{T}(\mathsf{P}); \mathit{tmp} := d; d := (\mathit{tmp}, \star_l^i); N := b)(\hat{A}) \tag{16}$$

holds for all $\hat{B} \subseteq \mathbf{Var}'$ and $\hat{A} \in \mathbf{AD}(\mathbf{Var}')$ satisfying $\xi(\hat{B}) \subseteq \hat{A}$.

If $A \in \mathbf{AD}(\mathbf{Var}')$ is the largest set satisfying

$$A = A' \cap \mathcal{A}(\mathcal{T}(\mathsf{P}); \mathit{tmp} := d; d := (\mathit{tmp}, \star_l^i); N := b)(A)$$

and $B \subseteq \mathbf{Var}'$ is the largest set satisfying

$$B = B' \cap \mathcal{B}(\mathsf{P}; N := b)(B)$$

(see the definitions of $\mathcal{A}$ and $\mathcal{B}$) then $\xi(B) \subseteq A$ by $\xi(B') \subseteq A'$ and (16). If $z \notin \mathbf{Var}_{\mathrm{asgn}}$ then we immediately have $z \in B_\bullet$ implying $(\mathbf{Var}_{\mathrm{T}} \cup \{d\}, z) \in A_\bullet$. If $z \in \mathbf{Var}_{\mathrm{asgn}}$ and $z \in B_\bullet$ then we have $z \in B$ and $N \in B$, implying $(\mathbf{Var}_{\mathrm{T}} \cup \{d\}, z) \in A$ and $(\mathbf{Var}_{\mathrm{T}} \cup \{d\}, N) \in A$. We need exactly these two pairs for $(\mathbf{Var}_{\mathrm{T}} \cup \{d\}, z) \in A_\bullet$ because $(\mathbf{Var}_{\mathrm{T}} \cup \{d\}) \backslash (\mathbf{Var}_{\mathrm{asgn}} \backslash \{d\}) = \mathbf{Var}_{\mathrm{T}} \cup \{d\}$.

8.1 Discussion

We have presented a program transformation that explains, how much information *declassify*-statements potentially leak. Actually, the presented transformation gives an upper bound to the potentially leaked information — they leak no more than what is deducible from the final value of the variable d. It is important to make the transformation as tight as possible — to write into the variable d as little as possible. The inclusion (7) still has to hold, though. Tighter transformation gives us more information / intuition about the exact amount of information leaked by *declassify*-statements.

The amount of leaked information according to our transformation has in some cases quite interesting characteristics. If the execution of the *declassify*-statement is not controlled by any *if*- or *while*-statement, then just the value of the declassified variable is leaked; this is straightforward. If the *declassify*-statement is under control of some *if*-statement then the declassifications before and after entering the branches of that *if*-statement are separated, but it is not explicitly recorded whether true- or *false*-branch was taken, nor is the exit from the branches recorded (the taken branch may sometimes be deduced from the value of the variable d through other means; for example, if the number of *declassify*-statements is different in the true- and false-branch, or if the declassified values in different branches look different).

If the *declassify*-statement is under the control of some *while*-statement then the things look a bit different. In this case we also record the beginning of the *while*-loop, but we also record the beginning of each iteration. The recording of the end of the loop is somewhat less explicit, the last $\star_l^i$ in the value of d denotes the end of the last iteration, but if we observe the value of d while the program has not yet terminated (and we do not know, which program point we are in), then we in general do not know whether the loop has finished the execution or not. So there seems to be some discrepancy between declassifications in *if*- and *while*-statements. Whether it is real or merely an artefact of our transformation, remains to be seen.

9 Conclusions

We have presented the notion of relatively secure information flow (together with a program analysis for it) and have used this notion to give a semantic meaning to *declassify*-statements. There are other potential uses of this notion that we have not investigated yet. For example, relative secrecy might be used for *modularizing* the security analyses or security proofs for programs. If a more precise analysis is necessary to verify the security of a program (for example, if the program contains cryptographic primitives; or if small leaks of confidential information are acceptable) then it may possible to use the described simple analysis for relative secrecy to verify that the program is secure if certain operations preserve its security, and then analyse these operations and the data flows leading to them with a more precise (and likely more complex) analysis. In this way, the more complex analysis does not have to be applied to the whole program but only to the parts needing the extra precision.

The modularization of arguments for secure information flow may be even more useful when we are adding a proof of secure information flow to a piece of mobile code. The proof could basically be the precomputed analysis result (with sufficiently many intermediate values to make its checking simple and straightforward); but if a simple Denning-style analysis is not sufficient to prove the security of information flow in our piece of code, then such proof may become rather large. Modularization allows us to use the extended expressiveness only where really necessary.

References

[1] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, 1976.

[2] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, 1977.

[3] Peeter Laud. *Computationally Secure Information Flow*. PhD thesis, Universität des Saarlandes, 2002.

[4] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, January 1999.

[5] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[6] Claude E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27:379–423 and 623–656, 1948.

[7] Dennis M. Volpano. Secure Introduction of One-way Functions. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW'00)*, pages 246–254, Cambridge, England, July 2000. IEEE Computer Society Press.

[8] Dennis M. Volpano and Geoffrey Smith. Verifying Secrets and Relative Secrecy. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 268–276, Boston, Massachusetts, January 2000. ACM Press.

[9] Dennis M. Volpano, Geoffrey Smith, and Cynthia Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.

[10] Steve Zdancewic. A type system for robust declassification. In *Proceedings of the 19th Conference on the Mathematical Foundations of Programming Semantics*, Electronic Notes in Theoretical Computer Science. Elsevier, March 2003.

[11] Steve Zdancewic and Andrew C. Myers. Robust Declassification. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 15–23, Cape Breton, Nova Scotia, June 2001. IEEE Computer Society Press.