An Information Flow Security Logic for Permission-Based Declassification Strategy

Zhenheng Dong[1], Yongxin Zhao[1*] and Qiang Wang[2]
[1] Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
[2] Chinese Academy of Military Science, Beijing, China

***Abstract*—With the increasing popularity of smartphones and the rapid development of mobile network, ensuring the security of mobile applications becomes more and more important, which has received substantial attention from both academia and industry. Information flow security, as a prominent approach to system and network security, aims at ensuring high security level information would not be accessed by analyzing the information with lower security levels. In this paper, we design a novel information flow security logic to reason about the security of mobile applications, leveraging on the idea of permission based declassification. Firstly, we propose a formal language with permission check branches, through which the access to the confidential information can be controlled. Then we present our novel information flow security logic based on the permission based declassification strategy, which can make the reasoning more precise by degrading the security level of the specific information. Finally, we demonstrate the usability of our logic via examples.**

***Index Terms*—Information Flow Security, Formal Language and Logic, Permission-Based Declassification, Mobile Applications**

I. Introduction

Information flow security is a prominent approach to system and network security. Given the fact that system components can be classified into different security levels, among which confidential information (e.g., private data resource) is marked as high security level, whereas public information (e.g., public program variable) is marked as low security level, the basic idea of information flow security is to ensure confidential information with high security level can not be obtained by analyzing the observable information with lower security levels. In the past decades, information flow security has been investigated and applied to various fields, e.g., operating systems [1], Web services [2] and cloud computing [3].

Among all the approaches to guarantee the information flow security, formal methods have been widely used and can be roughly divided into three categories, i.e., process algebra based methods, type system based methods and semantic based methods [4]. The work in [5] described four approaches to verifying security protocols based on process algebra. In [6] the authors used type system to check semantic errors in programs and to catch errors in code before program execution. In [7] the authors implemented a secure type system implementing integrity information flow control. The basic idea of semantic based methods is to bind a security level to program variables, and to adjust the security level according to the semantics of program operations dynamically. If there is no information leakage from the high security level to the low security level, the program access to the system is considered to be secure. The work in [8] proposed a secure semantic web service model by analyzing the web service security. In [9] the COVERN logic was proposed based on the lock mechanism for the security analysis of shared memory in concurrent programs. In [10] a non-blocking algorithm was proposed to avoid the use of locks on shared variables and data structures. Inspired by these previous work, we adopt the semantic based method to solve the information flow security in mobile applications.

In this paper, we propose a semantic-based approach to the information flow security among sequential mobile applications. Particularly, inspired by the literature [11] and [12], our approach introduces a permission-based declassification strategy to ensure that information will only be legally accessed by applications who have been granted the permissions. Throught the approach, the information can be downgraded from the origianl high security level to make the logical reasoning more accurate. The contributions of this work are as follows:

- **Permission-based formal language for information flow security:** The language abstracts application programs into functions, and uses function calls to represent information interactions among applications. Permissions are the basis for whether messages can be obtained in information interaction. In addition, the operational semantics of the language are strictly formulated, which is helpful for the formulation of reasoning logic.
- **Logic rules supporting declassification strategy:** This is the first semantic-based permission-dependent information flow security method, which adds the content of permission into the traditional information flow logic. In addition, the logic also adds a declassification strategy to make the reasoning more precise.

The rest of this paper is organised as follows. In Section II, we provide some running examples to illustrate the intuitive idea of our approach. A permission-based formal language is presented in Section III and its semantics is given in Section IV. In Section V, we discuss our novel informal flow security logic. Section VI illustrates the usability of our logic via examples and Section VII concludes the paper.

*Corresponding author: yxzhao@sei.ecnu.edu.cn
DOI reference number: 10.18293/SEKE2022-134

II. Running Examples

Before introducing the language and logic, we informally describe its core ideas and practical problems that can be solved through some examples.

The current popular definition of information flow security is non-interference [13], that is, if only the input changes, the attacker cannot observe the difference in execution results [14]. However, for mobile applications, traditional information flow security is not applicable due to the need for nested calls of programs, as shown in the example in Table I.

TABLE I: An example of traditional information flows

```
String getInformation(String name){
        String information;
        if(verifyPass(READ_PASSWORD))
                information=name.information;
        else
                information="";
        return information;
}
```

We can think of this example as a login service for social APPs, such as WeChat, instagram, etc., where the parameter $name$ is the username, and the function $verifyPass()$ is for authorization verification. If the password verification is passed, then the information $name.information$ of the user is returned. Otherwise, an empty string is returned. It is obvious that the user information is confidential and its security level should be high, while the security level of an empty string should be low. Because the traditional information flow security requires the two branches of the conditional statement to have the same security level, the non-interference can be guaranteed. At this time, if the information level is high, then the information can be used by other applications as the return value, and there will be a great security risk. Conversely, if the information level is low, either the application cannot obtain useful information, or there is an information leakage problem of assigning a high security level to a variable with a low security level. Therefore, a new strategy is needed to solve the resource acquisition problem of the application.

To solve this problem, the work in [15] proposed a type system that incorporates permissions in function types. And another work in [16] also proposed a type system that solves the problem of typing non-monotonic policies without resorting to downgrading or declassifying the information. Inspired by the above research, we define a statement that supports declassification assignment, as shown in Table II.

TABLE II: An example of our method

```
A.funA(name){
        init information=0;
        in{
                check(p){
                        then  information:=↓ name.information;
                        else  information:=0;
                }
                return information;
        }
}
```

We introduce a permission access control mechanism, and use the Check statement to replace the If-condition in the traditional information flow. If the application contains the permission p, then we can lower the security level by declassifying the assignment statement of the high security level $name.information$, and assignthe declassified variable to information. This can ensure that the return value of each function is a low security level, which ensures the security of the information flow of the system, and also solves the problem of application resource acquisition.

III. The Formal language

In this section, we present a formal language for a permission-based approach to information flow security.

The syntax of expressions is given below:

$$e ::= v \mid x \mid e \ op \ e \mid \overline{e}$$

Where v represents an integer value, x represents a variable, op is a binary operator defined on two expressions, $\overline{e} = [e_1, e_2, ..., e_n]$ represents an expression tuple, and each element in the tuple is an expression respectively.

The syntax of the command statement is given in the following:

$$\begin{aligned} c ::= \quad & skip \mid x := e \mid c_1; c_2 \mid if\ e\ then\ c_1\ else\ c_2 \\ & \mid while\ e\ do\ c \mid init\ x = e\ in\ c \mid x := call\ A.f(e) \\ & \mid x :=\downarrow e \mid check(p)\ then\ c_1\ else\ c_2 \end{aligned}$$

Where $skip$ means the empty statement does nothing. $init\ x := e\ in\ c$ is the definition statement of a local variable x, and the program block c represents the scope of the local variable x. $x := call\ A.f(e)$ means to call the function $f(e)$ of the application A, the expression e is the incoming parameter of the function, and uses the variable x receives the return value of the function call. $x :=\downarrow e$ represents the declassification assignment statement, which means that the expression e with high security level is allowed to downgrade the security level and assign it to the variable x. $check(p)\ then\ c_1\ else\ c_2$ represents the permission check statement, check whether the atomic permission p is included in the permission context, if it does, execute the statement c_1, otherwise execute c_2. Also, other statements are not much different from those of other programming languages.

A function definition is in the following:

$$F ::= A.f(\overline{input})\{init\ output = 0\ in\ \{c\ ;\ return\ output\}\}$$

Where $A.f$ denotes the function of application A whose function name is f. $\overline{input}$ is the formal parameter of the function, c is the execution statement of the function body, $output$ is the local variable and the return value of the function, and $\{c; return\ output\}$ is its scope. We only consider the closed function in this language, that is the variables that appear in c will only be variables introduced into the parameter input or local variables within the function.

IV. Operational semantics

A. Semantic Model

In the semantic model, we define a system state μ as a tuple $\langle mem, tr\rangle$, where mem represents memory and tr represents event trace.

The memory mem_μ denotes a set of assignments for all variables in the state μ, that is $mem_\mu = [x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots, x_n \mapsto v_n]$, where $x_1, x_2, \ldots, x_n$ represents a finite set of all variables in the system, and $mem_\mu(x_1)$ represents the value of the variable x_1 under the state of the system μ.

We use a sequence tr_μ to save all the assignment events from the initial state to the current state of the system. In this study, the event $event$ consists of two kinds of events: First, the ordinary assignment event $ASG\langle lvl, x, e\rangle$, which means to assign the value of the expression e to the variable x whose security level lvl. Second, the classification assignment event $DCF\langle lvl, x, e\rangle$ means that the value of the expression e is declassified and assigned to the variable x, and after declassification, the security level of variable x is lvl. The sequence of each historical event is called a trace, which is a finite collection of events.

B. Operational Semantics

We refer to [17] and [18] for our semantics. Our semantics is divided into two levels: expressions and statements.

The evaluation of the expressions are given in Table III. The semantic judgment of the expression has the form of $\mu \vdash e \Rightarrow v$, where μ represents the system state, e is the expression, and v is the value of the expression, which can be interpreted as in the state μ, the value of the expression e is v.

TABLE III: Evaluation of Expressions

$$\textbf{(Value)}\ \frac{}{\mu \vdash v \Rightarrow v}$$

$$\textbf{(Variable)}\ \frac{}{\mu \vdash x \Rightarrow mem_\mu(x)}$$

$$\textbf{(Tuple)}\ \frac{\mu \vdash e_i \Rightarrow v_i\ (1 \le i \le n)}{\mu \vdash \overline{e} \Rightarrow \overline{v}}$$

$$\textbf{(BiOp)}\ \frac{\mu \vdash e_1 \Rightarrow v_1 \quad \mu \vdash e_2 \Rightarrow v_2 \quad v_1\ op\ v_2 = v_3}{\mu \vdash e_1\ op\ e_2 \Rightarrow v_3}$$

(Value) indicates that in the system state μ, the value of the numerical expression v is v.

(Variable) indicates that in the system state μ, the value of the variable x is the value of x in the memory under the system state, which can be expressed as $mem_\mu(x_1)$.

(Tuple) indicates that in the system state μ, when $1 \le i \le n$, the value of the expression e_i is v_i respectively, then the value of the tuple expression $\overline{e}$ is expressed as $\overline{v}$.

(BiOp) indicates that in the system state μ, if the value of e_1 is v_1, the value of e_2 is v_2, and the value of v_1 and v_2 calculated by the binary operator is v_3. Then in this state, the result of the binary operation between $e1$ and $e2$ is $v3$.

The operational semantics of command statements are given in Table IV. The operational semantic judgment of the command statement has the form of $A, P \vdash (\langle mem, tr\rangle, c) \rightarrow \langle mem', tr'\rangle$. Where A represents the application, and the permission set P denotes the permission context, that is the set of permissions of the application which invokes the function of A. $\langle mem, tr\rangle$ represents a system state μ, tr represents the event execution sequence. This judgment means that in the environment of the application A with the permission context P, the system state before executing the statement c is $\langle mem, tr\rangle$, and after executing, it becomes $\langle mem', tr'\rangle$.

(Skip) and **(Seq)** are relatively simple, and they are not very different from the operational semantics of general sequential languages, so we will not go into details here. **(If)** and **(While)** use the expression e to determine whether the condition is true. For simplicity, we use TRUE and FALSE as the distinguishing criterion.

Two assignment statements correspond to two kinds of events in the system, that are ASG and DCF. When executing a normal assignment statement **(AssignN)**, we should add ASG event after the current trace tr. However, when executing the declassification assignment statement **(AssignD)**, we add DCF event after tr. Depending on the trace of the event history, we can distinguish which type of assignment statement is currently executing.

The local variable definition statement **(DefL)** means to define a local variable x whose initial value is the value v of the expression e. The scope of the variable x is only within the statement block c. Its operational semantics are to assign the value v of the expression e to the variable x, and execute the statement c under this premise. After execution ends, local variables should be removed from the memory. $mem' - x$ means to delete the local variable x from memory mem'.

The permission check statement **(Check)** is similar to an **(If)** statement. Where p is an atomic permission in the system permission set. Which branch of the statement is executed depends on the relationship of the atomic permission p to the permission context P. If P contains permission p, that is, $p \in P$, execute the statement c_1, otherwise execute c_2.

(Call) is more complex than the operational semantics of other statements. When application A calls function f of application B, first we need to find function f in all functions of application B. Then we use P_A to present the permission set of the application A, after that we give permission set P_A to application B, and execute function body in application B. Finally, assign the return value to the variable x. In short, application A calls the function of application B, we should check whether the corresponding permission exists in application A during the process of executing the function of B. This means that the permission set of application cannot be passed in recursive calls to functions.

V. The Logic

A. Access Control Model

In traditional information flow security, we introduce permissions for access control. We use $\mathcal{P}$ to denote a finite set of all permissions in the entire system, that is, $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$. P represents the permission context, and $P \subseteq \mathcal{P}$. We require

TABLE IV: Operational Semantics of Command Statements

$$\textbf{(Skip)}\ \frac{}{A,P\vdash(\langle mem,tr\rangle,skip)\rightarrow\langle mem,tr\rangle} \qquad \textbf{(AssignN)}\ \frac{\mu\vdash\ e\ \Rightarrow\ v \quad tr'=tr\cdot ASG}{A,P\vdash(\langle mem,tr\rangle,x:=e)\rightarrow\langle mem[x\mapsto v],tr'\rangle}$$

$$\textbf{(Seq)}\ \frac{A,P\vdash(\langle mem,tr\rangle,c_1)\rightarrow\langle mem',tr'\rangle \quad A,P\vdash(\langle mem',tr'\rangle,c_2)\rightarrow\langle mem'',tr''\rangle}{A,P\vdash(\langle mem,tr\rangle,c_1;c_2)\rightarrow\langle mem'',tr''\rangle}$$

$$\textbf{(IfF)}\ \frac{\mu\vdash e\Rightarrow FALSE \quad A,P\vdash(\langle mem,tr\rangle,c_2)\rightarrow\langle mem',tr'\rangle}{A,P\vdash(\langle mem,tr\rangle,if\ e\ then\ c_1\ else\ c_2)\rightarrow\langle mem',tr'\rangle} \qquad \textbf{(IfT)}\ \frac{\mu\vdash e\Rightarrow TRUE \quad A,P\vdash(\langle mem,tr\rangle,c_1)\rightarrow\langle mem',tr'\rangle}{A,P\vdash(\langle mem,tr\rangle,if\ e\ then\ c_1\ else\ c_2)\rightarrow\langle mem',tr'\rangle}$$

$$\textbf{(WhileF)}\ \frac{\mu\vdash e\Rightarrow FALSE}{A,P\vdash(\langle mem,tr\rangle,while\ e\ do\ c)\rightarrow\langle mem,tr\rangle}$$

$$\textbf{(WhileT)}\ \frac{\mu\ \vdash\ e\ \Rightarrow TRUE \quad A,P\vdash(\langle mem,tr\rangle,c)\rightarrow\langle mem',tr'\rangle \quad A,P\vdash(\langle mem',tr'\rangle,while\ e\ do\ c)\rightarrow\langle mem'',tr''\rangle}{A,P\vdash(\langle mem,tr\rangle,while\ e\ do\ c)\rightarrow\langle mem'',tr''\rangle}$$

$$\textbf{(DefL)}\ \frac{\mu\ \vdash\ e\ \Rightarrow v \quad A,P\vdash(\langle mem[x\mapsto v],tr\rangle,c)\rightarrow\langle mem',tr'\rangle}{A,P\ \vdash\ (\langle mem,tr\rangle,init\ x=e\ in\ c)\rightarrow\langle mem'-x,tr'\rangle} \qquad \textbf{(AssignD)}\ \frac{\mu\vdash\ e\ \Rightarrow\ v \quad tr'=tr\cdot DCF}{A,P\vdash(\langle mem,tr\rangle,x:=\downarrow e)\rightarrow\langle mem[x\mapsto v],tr'\rangle}$$

$$\textbf{(CheckF)}\ \frac{p\notin P \quad A,P\vdash(\langle mem,tr\rangle,c_2)\rightarrow\langle mem',tr'\rangle}{A,P\vdash(\langle mem,tr\rangle,check(p)\ then\ c_1\ else\ c_2)\rightarrow\langle mem',tr'\rangle} \qquad \textbf{(CheckT)}\ \frac{p\in P \quad A,P\vdash(\langle mem,tr\rangle,c_1)\rightarrow\langle mem',tr'\rangle}{A,P\vdash(\langle mem,tr\rangle,check(p)\ then\ c_1\ else\ c_2)\rightarrow\langle mem',tr'\rangle}$$

$$\textbf{(Call)}\ \frac{Find(B.f)=B.f(\overline{input})\{init\ output=0\ in\ \{c\ ;\ return\ \ output\}\} \quad \mu\vdash\overline{e}\Rightarrow\overline{v} \quad B,P_A\vdash(\langle mem[\overline{input}\mapsto\overline{v},output\mapsto 0],tr\rangle,c)\rightarrow\langle mem',tr'\rangle}{A,P\vdash(\langle mem,tr\rangle,x:=call\ B.f(\overline{e}))\rightarrow\langle mem[x\mapsto mem(output)],tr'\rangle}$$

that the permission set of each application is statically allocated in the initial state and cannot be modified dynamically.

We divide variables into two categories, one is the variable that requires permission to access, and the other is the variable that can be accessed without permission. For variables that require permission to access, we use the function $\Gamma(x)$ to represent the permission required to access the variable x. For example, when $\Gamma(x)=p$, it means permission p corresponds to the access to the variable x, then when the permission set P contains the permission p, the currently executing application can get access to the variable x through $check(p)$, otherwise the application cannot access the variable x. For variables that do not require permission access, it is not necessary to read the variables through permission control.

We assume that the application running at this time is A. In the case of no function call, the context permission set P is the permission set P_A of the application A. The application A can access all variables that do not require permission access and the variables corresponding to the permissions possessed by its permission set P_A. When making a function call, if the application A calls the function of the application B, the context permission set P is the permission set P_A, and then we use P to execute application B.

B. Information Flow Model

We define the security level lvl for all expressions on the grid $Low\leq lvl\leq High$. First, we define the security level for the expression e. We stipulate that the numeric type v has no specific security level, and its security level depends on the security level of the variable it is assigned to. For the variable x that has defined the permission function $\Gamma(x)$, we require its security level to be the highest $High$. For some variables, we use the function $L(x)$ to define the security level, and use $L(x)$ to represent the highest security level of the data that the variable x can hold at any time. This means that, in all assignments, the security level of the variable x cannot exceed the upper limit of $L(x)$. For the variable x that does not define $L(x)$, we default its security level to any level. When we assign the value of the expression e to the variable x through $x:=e$, x automatically has the security level of expression e. The current security level of the variable x depends on the security level of the last assigned expression. For the binary operation expression $e:=e_1\ op\ e_2$, the security level of the expression e is the one with the higher security level among the two expressions involved in the operation. Similarly, in the expression tuple $e=\{e_1,e_2,e_3,\ldots,e_n\}$, the security level of the expression e is the highest security level held by all expressions in the tuple. And we use lvl_e to denote the current security level of expression e.

We introduce the attacker role to prove the security of the system, we assume that the attacker is a passive attacker who can only get information by observing the execution of the program. Specifically, we assume that the attacker has an attack level of lvl_a, then for all variables with a security level of $lvl\leq lvl_a$, the attacker can observe them. At the same time, if the assignment operation is declassification assignment, the variable after declassification which security level is $lvl\leq lvl_a$ can also be observed by the attacker.

Our logic has judgements of the form $\mu,A,P,lvl_a\ \vdash\ c$, where μ represents the current system state, A represents the name of the application, P represents the permission context, lvl_a represents the security level of the attacker, and c represents a program statement. This judgment is true if and only if the system state is μ, and we execute application A on permission context P, the program text c will not leak information to the lvl_a level.

C. Declassification Model

In order to make the declassification assignment statement be executed safely, we define a declassification predicate:

$$D(lvl_{src},lvl_{des},\mu,c,P).$$

In the definition of the predicate, lvl_{src} represents the security level of the expression before declassification, lvl_{des}

TABLE V: Rules of the logic

(**R-Skip**) $\dfrac{}{\mu, A, P, lvl_a \vdash skip}$ (**R-UAsgN**) $\dfrac{L(x)\ \ is\ \ undefined}{\mu, A, P, lvl_a \vdash x := e}$

(**R-LAsgN**) $\dfrac{lvl_e \leq L(x)}{\mu, A, P, lvl_a \vdash x := e}$ (**R-Seq**) $\dfrac{\mu, A, P, lvl_a \vdash c_1 \quad A, P \vdash (\mu, c_1) \rightarrow \mu' \quad \mu', A, P, lvl_a \vdash c_2}{\mu, A, P, lvl_a \vdash c_1; c_2}$

(**R-IfF**) $\dfrac{\mu \vdash e \Rightarrow FALSE \quad \mu, A, P, lvl_a \vdash c_2 \quad lvl_e \leq lvl_a}{\mu, A, P, lvl_a \vdash if\ e\ then\ c_1\ else\ c_2}$ (**R-IfT**) $\dfrac{\mu \vdash e \Rightarrow TRUE \quad \mu, A, P, lvl_a \vdash c_1 \quad lvl_e \leq lvl_a}{\mu, A, P, lvl_a \vdash if\ e\ then\ c_1\ else\ c_2}$

(**R-WhileF**) $\dfrac{\mu \vdash e \Rightarrow FALSE \quad lvl_e \leq lvl_a}{\mu, A, P, lvl_a \vdash while\ e\ do\ c}$

(**R-WhileT**) $\dfrac{\mu \vdash e \Rightarrow TRUE \quad \mu, A, P, lvl_a \vdash c\ ;\ while\ e\ do\ c \quad lvl_e \leq lvl_a}{\mu, A, P, lvl_a \vdash while\ e\ do\ c}$

(**R-DefL**) $\dfrac{\mu \vdash e \Rightarrow v \quad \langle mem[x \mapsto v], tr\rangle, A, P, lvl_a \vdash c}{\mu, A, P, lvl_a \vdash init\ x = \ e\ in\ c}$ (**R-AssignD**) $\dfrac{D(lvl_e, L(x), \mu, x :=\downarrow e, P)}{\mu, A, P, lvl_a \vdash x :=\downarrow e}$

(**R-CheckF**) $\dfrac{p \notin P \quad \mu, A, P, lvl_a \vdash c_2}{\mu, A, P, lvl_a \vdash check(p)\ then\ c_1\ else\ c_2}$ (**R-CheckT**) $\dfrac{p \in P \quad \mu, A, P, lvl_a \vdash c_1}{\mu, A, P, lvl_a \vdash check(p)\ then\ c_1\ else\ c_2}$

(**R-Call**) $\dfrac{Find(B.f) = B.f(\overline{input})\{init\ output = 0\ in\ \{c\ ;\ return\ \ output\}\} \quad \mu \vdash \overline{e} \Rightarrow \overline{v} \quad \langle mem[\overline{input} \mapsto \overline{v}, output \mapsto 0], tr\rangle, B, P_A, lvl_a \vdash c \quad lvl_{output} \leq L(x)}{\mu, A, P, lvl_a \vdash\ x := call\ B.f(\overline{e})}$

represents the security level of the expression after declassification, μ represents the current system state, c represents program statement and P represents the permission context. For example, when the system state is μ, a declassification statement $x :=\downarrow e$ is executed on the permission context P. At this time, its declassification predicate is $D(lvl_e, L(x), \mu, x :=\downarrow e, P)$.

Whether declassification predicate is hold depends on the permission function $\Gamma(e)$ of the declassification expression e. If the permission function $\Gamma(e)$ of the expression e is defined and $\Gamma(e) \subseteq P$, then the declassification predicate holds, and the declassification operation is secure at this time. Otherwise, either when the permission function $\Gamma(e)$ of the expression e is not defined, or the permission function is defined, but $\Gamma(e) \not\subseteq P$, the declassification predicate does not hold, and the declassification operation is insecure at this time.

D. Rules

Our proposed logic rules are shown in the table V.

Some of these rules, such as **R-Skip** and **R-Seq** statements are relatively simple, and we can be easily analogized to Hoare logic. For assignment statements, we can divide them into two cases **R-UAsgN** and **R-LAsgN** according to whether the variable defines the security level function L. If a variable x does not define the security level function $L(x)$, it means that the variable can receive the value of the expression of all security levels, that is, its security level is the highest. Therefore, in this case, it is secure to assign any expression to the variable x. When the security level function $L(x)$ of the variable x is defined, it means that the variable x can contain a security level that cannot exceed $L(x)$. At this time, the assignment statement needs to guarantee $lvl_e \leq L(x)$, otherwise we think that the information flow is insecure.

The conditional statement **R-If** is similar to the loop statement **R-While**. In order to prevent the attacker from obtaining information of high security level in the condition through different running results, we require that the security level of the attacker is not lower than the security level of the condition, that is $lvl_e \leq lvl_a$.

Local variable definition statement **R-DefL**, since this statement is not an assignment statement, we have no requirements for the definition of the variable, we only need to ensure that the variable is secure for information flow within its scope. The permission check statement **R-Check**, we divide the statement into two cases according to whether the atomic permission p belongs to the permission set P. Each situation corresponds to two different permission sets P. In addition, we also need to ensure that the information flow of the program is secure in each branch of the permission check.

Because we defined the declassification predicate D in the previous section, the declassification assignment statement **R-AsgD** is secure only if the declassification predicate holds, otherwise it is not. For the function call statement **R-Call**, we need to satisfy the information flow security inside the function body and the function return value respectively.

VI. Verification

In this section we illustrate the usefulness of our proposed logic through an example. The example code is shown in Table VI below.

We can think of this example as a mobile banking login application. When we log into the mobile banking, we must first perform mobile code verification on our mobile phone, and then we need to identify the person before we can enter the bank account. We can assume that the above code simulates this function, where the permission p_1 indicates whether the user has the permission to verify the mobile phone code verification, and the permission p indicates the permission of the identity verification. Obviously, these two permissions are indispensable, otherwise we will not be able to login in normally.

TABLE VI: An application example

```
B.funB(){
   init y=0;
   in{
     check(p1){
       then  y:=call A.fun();
⟨mem[y ↦ 0], tr⟩, B, P_B, lvl_a ⊢ y := callA.fun()
       else  y:=0;
     }
⟨mem[y ↦ 0], tr⟩, B, P_B, lvl_a ⊢ check(p1) then c1 else c2
     return y;
   }
μ, B, P_B, lvl_a ⊢ init y = 0 in c
}
A.fun(){
   init x=0;
   in{
     check(p){
       then  x:=↓ information;
⟨mem[y ↦ 0, x ↦ 0], tr⟩, A, P_B, lvl_a ⊢ x :=↓ information
       else  x:=0;
     }
⟨mem[y ↦ 0, x ↦ 0], tr⟩, A, P_B, lvl_a ⊢ check(p) then c1 else c2
     return x;
   }
⟨mem[y ↦ 0], tr⟩, A, P_B, lvl_a ⊢ init x = 0 in c
}
```

In this example, we assume that the permission set P_B of the application B contains the permissions p and p_1, then in the process of executing the application B, the permission context P is P_B. First we define a local variable y. Then execute the $check(p_1)$ statement. Because the permission set of the application B contains the permission p_1, the $then$ branch is executed to enter the function call statement. In the function call statement, we should use the permission set P_B of the application B as the permission context into the application A for execution. After defining the local variable x in A, enter the $check(p)$ statement. At this time, because the permission set P_B also contains the permission p, we use declassification assignment statement assigns the high security level $information$ to the variable x with the reduced security level, and returns it through the return value. After returning to the application B, we use the variable y to receive the return value of the function call, and get the final bank account information, and the login is successful.

In information flow security, the use of local variables needs to ensure that the information flow is secure in its scope, and the function call needs to ensure that the information flow is secure in the function body. Therefore, if and only if the security level of the return value of the function is less than or equal to $L(y)$ and the classification predicate $D(lvl_{information}, L(x), \mu, x :=\downarrow information, P)$ holds, the information flow is secure, otherwise it is insecure.

VII. Conclusion and Future Work

In this work, We present a formal language and the corresponding logical rules for proving the information flow security of mobile applications. Our approach has well defined semantics and makes use of a permission based declassification strategy, which makes the reasoning more accurate.

In the future, we would like to extend the access control policy to consider solutions to the branching problem that relies on secrets. In addition, we will also extend the semantics and the logic to handle the problem of non-monotonic of permissions.

Acknowledgements

This work is supported by Shanghai Science and Technology Commission Program under Grant 20511106002, Shanghai Trusted Industry Internet Software Collaborative Innovation Center and the Fundamental Research Funds for the Central Universities.

References

[1] M.Krohn and E.Tromer, "Noninterference for a practical difc-based operating system," in *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, 2009, pp. 61–76.

[2] N. B. Said and L. Cristescu, "End-to-end information flow security for web services orchestration," *Science of Computer Programming,187:102376*, 2020.

[3] J. Bacon, D. Eyers, T. Pasquier, J. Singh, L. Papagiannis, and P. Piezuch, "Information flow control for secure cloud computing," in *IEEE Transactions on Network and Service Management*, vol. 11, no. 1, 2014, pp. 76–89.

[4] "Review and prospect for information flow security technology," in *Journal of Nanjing University of Posts and Telecommunications*, vol. 31, no. 5, 2011.

[5] L. Mengjjun, L. Zhoujun, and C. Huowang, "A survey of security protocol verification base on process algebras," in *Journal of Computer Reserach and Development*, vol. 41, no. 7, 2004, pp. 1097–1103.

[6] D. Zhiyi, S. Guoxin, and S. Zhiqing, "Type system and the correctness of program," in *Computer Science*, vol. 33, no. 1, 2006, pp. 141–143.

[7] W. Libin, "Information flow control for integrity based on type system," in *Journal of South China Normal University*, vol. 3, 2006, pp. 42–47.

[8] L. Chengcheng, Z. Yongsheng, and L. Guangyu, "Research on a secure semantic web services mode," in *Computer Technology and Development*, vol. 20, no. 2, 2010, pp. 170–174.

[9] T. Murray, R. Sison, and K. Engelhardt, "Covern: A logic for compositional verification of information flow control," in *2018 IEEE European Symposium on Security and Privacy*, 2018, pp. 16–30.

[10] N. Coughlin and G. Smith, "Rely/guarantee reasoning for noninterference in non-blocking algorithms," in *2020 IEEE 33rd Computer Security Foundations Symposium(CSF)*, 2020, pp. 16–30.

[11] L. Hao, L. Qiang, Y. Jiwen, and Q. Peide, "A security system model based on mandatory access control and information flow," in *Computer Engineering and Science*, vol. 27, no. 3, 2005, pp. 16–20.

[12] D. Schoepe, T. Murray, and A. Sabelfeld, "Veronica:expressive and precise concurrent information flow security," in *IEEE 33rd Computer Security Foundation Symposium(CSF)*, 2020.

[13] J.Goguen and J.Mesegue, "Security policies and security models," in *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, 1982.

[14] R. Giacobazzi and I. Mastroeni, "Abstract non-interference: A unifying framework for weakening information-flow," in *ACM Transactions on Privacy and Security*, vol. 21, no. 2, 2018.

[15] A.Banerjee and D.A.Naumann, "Stack-based access control and secure information flow," in *Journal of Functional Prpgramming*, vol. 15, no. 2, 2005, pp. 131–177.

[16] H. Chen, A. Tiu, Z. Xu, and Y. Liu, "A permission-dependent type system for secure information flow analysis," in *IEEE 31st Computer Security Foundations Symposium*, 2018, pp. 218–232.

[17] Y. Zhao, X. Wu, J. Liu, and Y. Yang, "Formal modeling and security analysis for openflow-based networks," in *International Conference on Engineering of Complex Computer Systems*, 2018, pp. 201–204.

[18] Y. Zhao, X. Zhang, L. Shi, G. Zeng, F. Sheng, and S. Liu, "Towards a formal approach to defining and computing the complexity of component based software," in *Asia-Pacific Software Engineering Conference*, 2019, pp. 331–338.