BSPL: A Language to Specify and Compose Fine-grained Information Flow Policies

Stéphane Geller, Valérie Viet Triem Tong, Ludovic Mé
Team Cidre
SUPELEC
Rennes, France
Email: stephane.geller@supelec.fr, valerie.viettriemtong@supelec.fr, ludovic.me@supelec.fr

***Abstract*—We tackle the problem of operating systems security. The seriousness of the vulnerabilities in today's software underlines the importance of using a monitor at the operating system level to check the legality of operations executed by (un)trusted software. Information flow control is one way to track the propagation of information in order to raise an alert when a suspicious flow, consequence of an attack, occurs. We propose here BSPL, a language to specify a fine-grained information flow policy. We present how BSPL enables to precisely specify the expected behavior of applications relatively to sensitive pieces of information. We also propose a way to compose such information flow policies.**

Keywords*—*Information flow policies, specification language, composition of information flow policies

I. INTRODUCTION

Information flows characterize all the operations on the system resulting in data movements from one location to another, such as when a process reads or writes to a file.

Over the past decade, many worksd [1], [2], [3], [4], [5] have focused on providing strong information security guarantees by tracking and/or controlling information flows.

Basically, information flow tracking identifies sensitive data with a dedicated mark called taint, label or tag. The marks are propagated along the flow to taint objects in the system and a monitor uses the marks to check the legality of the information flows, with respect to a previously defined "information flow policy".

We believe that there is a need for a language to precisely define such policies, as proposed by Efstathopoulos and Kohler, who were the first to present such a language [6].

In this paper, we focus on the `Blare` monitor [7]. `Blare` monitors information flows at the operating system level. It uses two marks (called tags) associated with any container of information in the operating system. This monitor has shown to be helpful to detect attacks, especially those that generate data leakage [8]. Though, as with other existing monitors, `Blare` suffers from a major drawback: the information flow policy that manages the monitor is either a toy policy [2], set by hand [9], or automatically computed from an access control policy [8].

In this article, we introduce the `Blare` Security Policy Language (BSPL). This language has been created to be a convenient way to specify an information flow security policy, in particular in contexts where several types of data coexist in the same system, like on a smartphone. BSPL allows a high level expression of the policy while keeping it simple and clear. A BSPL policy can be used to configure an information flow monitor (in our case `Blare`) to detect illegal read or write accesses to sensitive information by any malicious software.

On smartphones, the global information flow policy associated to the system can be viewed as a composition of multiple smaller policies provided by the application developers, the administrators and the users. This motivates our work around the composition of policies that is also presented in this paper.

The two major contributions of this paper are, thus: (1) the BSPL language and (2) the composition definition for policies written in this language. We also propose a consistency property for such information flow policies and prove that a policy constructed using BSPL is consistent and it is easy to prove that the composition of consistent policies is consistent.

The remainder of this paper is organized as follows: Section II presents the related work. Section III introduces our underlying model of information flow. In Section IV, we present the BSPL syntax and semantics, and Section V shows how to compose policies. Lastly, before concluding the paper, in Section VI, we illustrate the use of BSPL to express information flow policies.

II. RELATED WORK

As far as we know, the work most related to our approach has been proposed by Efstathopoulos and Kohler [6]. They proposes a policy description language where application developers can express their security requirements in terms of communication relationships. They claim that concentrating the policy specification in one place makes policies easier to write and reason about. They also provide a parser to translate these communication relationships considered as high-level policy to the equivalent Asbestos labels [10].

Unfortunately, they note that their language does not capture the full expressiveness of the Asbestos labels. More precisely, a process can be impacted by different policies but the language does not permit the construction of a global policy, which includes all these fragments of policy. In other words, their language does not support the composition of different policies concerning the same process. Our composition

 ISBN: 978-1-61208-298-1

mechanism could solve this type of problem. They also have trouble inferring the high-level policy from the labels of the system.

We want to pursue the efforts initiated in [6] to concentrate the policy specification in one place. However, contrarily to what has been proposed by Efstathopoulos, our high-level language allows such a centralized definition while being very close to the actual labels managed by the `Blare` monitor. Hence, it would be easy to rebuild the system global policy in BSPL given the different labels found in that system. In addition, we also propose a consistency property, which permits the elimination of ill-formed information flow policies.

In the following section, we detail the model of labels used by the `Blare` monitor and thus describe the requirements for our specification language, BSPL.

III. THE UNDERLYING MODEL OF INFORMATION FLOW

`Blare` is a model of information flow created to accomplish mainly two objectives: (1) the tracking of the content of the containers of the system and (2) the easy verification that this content respects our information flow policy. To accomplish these goals, we distinguish the containers from their content, potentially composed of multiple pieces of information depending on the time of the observation.

We separate containers of information into three categories: $\mathcal{PC}$ denotes the set of persistent containers (typically files), and $\mathcal{E}xecute(\mathcal{X})$ (or shortly $\mathcal{E}$) denotes the set of processes executing a code in $\mathcal{X}$. We also consider users as containers of information. We use $\mathcal{U}$ to denote the set of users. In the following we will thus use the notation $\mathcal{C} = \mathcal{PC} \cup \mathcal{U} \cup \mathcal{E}$ to deal with *any container of information*.

To differentiate between a piece of information which has been executed (a running code) and a piece of information which has not, we define two disjoint sets of meta-information $\mathcal{I}$ and $\mathcal{X}$ to track these pieces of information. The elements of the first set correspond to non-executed pieces of information, while those of the second set correspond to pieces of information executed by processes. Once a piece of information identified by a meta-information $\mathcal{I}$ is executed, the meta-information used to track it will be changed thanks to the function $running : \mathcal{I} \rightarrow \mathcal{X}$. For instance, if `app` is an application considered as sensitive, we attach $i \in \mathcal{I}$ to its code viewed as a data stored in a file; once `app` is launched by a process p we consider that p is running the code denoted by $x \in \mathcal{X}$ such that x is equal to $runnning(i)$. This distinction permits to distinguish the policy associated to a code viewed as data, and to the same code when it runs. The identifiers in $\mathcal{I}$ and $\mathcal{X}$ are either defined at the initialization of the model or during the execution of the system if new content is added.

Any container existing in the operating system is characterized with two tags: an *information tag* and a *policy tag*. The *information tag* is a collection of elements of $\mathcal{I} \cup \mathcal{X}$ that denotes the origins of the current content of the container (it is the tag used to accomplish the objective (1)). The policy tag is a set of elements of $\mathscr{P}(\mathcal{I} \cup \mathcal{X})$ that denotes the information flow policy attached to the container (it is the tag used to accomplish the objective (2)).

The information flow policy defines for each of these containers (let us remind that users are viewed as containers) which data mixture they are allowed to contain. The information flow policy uses the function $\mathbb{P}_{\mathcal{C}} : \mathcal{C} \rightarrow \mathscr{P}(\mathscr{P}(\mathcal{I} \cup \mathcal{X}))$ as building block. This function defines the data mixture allowed to flow into any container of information. For instance, $\mathbb{P}_{\mathcal{C}}(\mathtt{c}) = \{\{i_1, i_2, x_1\}, \{i_1, i_3, x_1\}\}$ means that a process executing the tainted program x_1 is allowed to write into the container `c`; it also means that `c` can either contain data computed from the tainted data i_1 and i_2 or data computed from i_1 and i_3.

When a process is launched during the execution, the value of its policy tag depends on the process owner and the policy tag attached to the code the process is currently running. More precisely, let us consider a process p_{new} resulting from the execution of the binary file `bf` by a process p. Let us also consider that p has an information tag equal to $\mathtt{I}_p \subseteq \mathcal{I} \cup \mathcal{X}$ and p executes a binary file `bf`, which has an information tag equal to $\mathtt{I}_{bf} \subseteq \mathcal{I} \cup \mathcal{X}$. The information tag $\mathtt{I}_p$ tells us what information is transmitted (in the worst case) by p to its child p_{new}. The information tag $\mathtt{I}_{bf}$ tells us which code will be executed in p_{new}. The information tag of p_{new} is thus equal to the union of the information tag of its father process and the function $running$ applied to the information tag of the executed binary file `bf`: $\mathtt{I}_p \cup running(\mathtt{I}_{bf})$. Its policy tag $\mathtt{P}_{p_{new}}$ depends on the policy tag $\mathtt{P}_u \subseteq \mathscr{P}(\mathcal{I} \cup \mathcal{X})$ of the process owner $u \in \mathcal{U}$ and on the value of $\mathbb{P}_{\mathcal{C}}(\mathcal{E}xecute(running(i)))$ for any i in $\mathtt{I}_{bf}$. $\mathtt{P}_u$ is defined by $\mathbb{P}_{\mathcal{C}}(u)$. Lastly, the policy tag $\mathtt{P}_{p_{new}}$ is defined by $\sqcap_{i \in \mathtt{I}_{bf} \setminus \mathcal{X}} \mathbb{P}_{\mathcal{C}}(\mathcal{E}xecute(running(i))) \sqcap \mathtt{P}_u$ ($A \sqcap B = \{a \cap b | a \in A, b \in B\}$).

The policy tag $\mathtt{P}_c$ of a container of information (whatever its type is) represents the set of combinations of sensitive pieces of information which can legally be found in the container. The fact that the content of a container c is legal in accordance with an information flow policy can be verified by checking if its information tag $\mathtt{I}_c$ is included in at least one element of its policy tag $\mathtt{P}_c$.

The function $\mathbb{P}_{\mathcal{C}}$ permits defining which data mixture is allowed to flow into any type of containers of information. This function implicitly defines a second function that associates to sensitive pieces of information i their possible mixes and locations. We denote this function $\mathbb{P}_{\mathcal{I} \cup \mathcal{X}}$, its type is $\mathcal{I} \cup \mathcal{X} \rightarrow \mathscr{P}(\mathcal{C} \times \mathscr{P}(\mathcal{I} \cup \mathcal{X}))$. This function associates to any identifier of sensitive piece of information $ix \in \mathcal{I} \cup \mathcal{X}$ a set of pairs (c, s) where c can be either a persistent container, a user or a process and s is a set of identifiers of pieces of information. A pair $(c, s) \in \mathbb{P}_{(\mathcal{I} \cup \mathcal{X})}(ix)$ means that a data computed from the piece of information ix is allowed to flow into the container (persistent, user or process) c mixed with any data computed from any subset of pieces of information appearing in s.

We have proved in [11] that this model of information flows was complete: if a violation of the policy occurs, it is detected. We can also prove that the model is sound. Nevertheless, this proof of soundness is less relevant, as it supposes that any information flow can be observed. This "total observability" is in practice impossible to obtain. This means that the implementation of our model is not exempt of false positives. Nevertheless, our experiments have shown that this false positive rate is lower with a monitor implementing our model than with more classical monitors.

 ISBN: 978-1-61208-298-1

In previous works [2], [9], we have proposed constructing an information flow policy in specifying for each container the list of data mixtures that the container is allowed to contain. In other words, the definition of the policy was guided by containers and we directly defined the policy through the specification of $\mathbb{P}_{\mathcal{C}}$. In this case, the function $\mathbb{P}_{\mathcal{I}\cup\mathcal{X}}$ is defined by reversing $\mathbb{P}_{\mathcal{C}}$. We believe that it is interesting to permit guiding the definition of an information flow policy by specifying where data mixtures are allowed to flow and thus in partially defining the function $\mathbb{P}_{\mathcal{I}\cup\mathcal{X}}$. BSPL offers such a feature as it is detailed in the following section.

If we permit defining an information flow policy through the definition (even partial) of the function $\mathbb{P}_{\mathcal{I}\cup\mathcal{X}}$ we may face an inconsistency problem of the policy. Intuitively, an information flow policy is consistent if when a piece of information is allowed to flow into a container with a data mixture, it is equivalent to two things : the concerned container is also allowed to receive this data mixture and all the pieces of information of the mixture are allowed to flow in the container with the same mixture. Definition 1 makes this notion clear.

Definition 1: An information flow policy is consistent if the following formulas are equivalent :
$\forall c \in \mathcal{C}, \forall s \in \mathscr{P}(\mathcal{I} \cup \mathcal{X})$,

$$\exists ix \in s, (c, s \setminus \{ix\}) \in \mathbb{P}_{\mathcal{I}\cup\mathcal{X}}(ix) \quad (1)$$
$$\forall ix \in s, (c, s \setminus \{ix\}) \in \mathbb{P}_{\mathcal{I}\cup\mathcal{X}}(ix) \quad (2)$$
$$s \in \mathbb{P}_{\mathcal{C}}(c) \quad (3)$$

When a policy is consistent, we can express the two parts of the policy in function of the other: $\mathbb{P}_{\mathcal{I}\cup\mathcal{X}}(ix) = \{(c, I) | c \in \mathcal{C} \wedge I \cup \{ix\} \in \mathbb{P}_{\mathcal{C}}(c)\}$ and $\mathbb{P}_{\mathcal{C}}(c) = \bigcup_{ix\in\mathcal{I}\cup\mathcal{X}} \{I \cup \{ix\} | (c, I) \in \mathbb{P}_{\mathcal{I}\cup\mathcal{X}}(ix)\}$. An information flow policy uniquely specified through one of the two is necessarily consistent since the last function is in this case entirely defined by reversing.

In the following section, we propose a dedicated language, BSPL, associated to this model.

IV. THE BSPL LANGUAGE

A. BSPL Grammar

A `Blare` policy is basically composed of two elements: the definition of sensitive data and their corresponding policy, and the definition of containers and their corresponding policy.

The first element on the policy is called *Data policy* and defines the policy attached to sensitive data. Pieces of information listed in this element belong to the user/the application/the system that defines the current policy. These pieces of information are thus considered as sensitive. This element is composed of a sequence of element `data` (each one characterizing a single piece of information, for example the initial content of a file), which is composed of:

- an element `data_identification` that permit the exact characterization of the sensitive piece of information (it gives its alias, its original container, its owner, its type (executable (X) or not (I)), the expected `reaction` of the monitor if it observes an illegal information flow: either it has to block the incriminated information flow or it has to raise an alert. Lastly an element `data_identification` has an attribute `case_of_unknown_containers` that is helpful for policy composition. This last attribute specifies the behaviour of the information identified above when the policy has to be composed with another policy that specifies information flows with this piece of information. It can be either `Ask` for *ask to the owner of the piece of information* (encouraged default value), `AlwaysAccept` (very permissive) or `NeverAccept` (very protective).

- a list of elements `can_flow` each of them having two attributes: `into` that denotes a container and `mixed_with` that denotes a list of sets of identifiers of sensitive data.

The attributes composing the element `can_flow` mean that the sensitive data identified by the previous element `identification` can be mixed with any set of pieces of information listed in `mixed_with` in the container in the attribute `into`. We can use the particular identifier `all_data`, which will mean all the possible sensitive data. The precise semantics associated with the group Data_policy is given in Section IV-B.

The second element defines the information flow policy attached to containers of information and is called *Containers_policy*. This element is composed of a list of element `container`. An element `container` is defined by:

- an element `container_identification` with five attributes that permit the exact characterization of the container (it gives an alias for the container, its original location, its owner and its type (file, process executing a code identified previously in `data_policy`) or user) , the expected `reaction` of the monitor if it observes an illegal information flow and lastly a behaviour. Dually to the case of unknown containers, an element `container_identification` has an attribute `case_of_unknown_data`, which specifies the behaviour of the container identified above when the policy has to be composed with another policy that specifies information flows with this container. It can be either `Ask` for *ask to the owner of the container* (encouraged default value), `AlwaysAccept` (very permissive) or `NeverAccept` (very protective).

- an element `can_receive` composed of one attribute: `mixture_of_data`, which indicates the mixture of data allowed to flow into the concerned container: `mixture_of_data` has to be a list of list of pieces of information separated with commas or it can be simply `all_data` (all the possible sensitive data).

In the next subsection, we describe the sense of the syntax given in this section, i.e., the semantics of BSPL, using the notations introduced in Section III.

 ISBN: 978-1-61208-298-1

B. BSPL semantics

We define two tables $\mathbb{T}_{\mathcal{I}\cup\mathcal{X}}$ and $\mathbb{T}_{\mathcal{C}}$. The first table $\mathbb{T}_{\mathcal{I}\cup\mathcal{X}}$ is used to store information permitting to characterize sensitive data (their owners, their initial location, their type (executable or not) and the reaction attached to an illegal flow with the data.. The second table $\mathbb{T}_{\mathcal{C}}$ is used to store the origin, the owner and the reaction attached to a container defined in the policy.

The monitor refers to these tables when it detects an illegal information flow, since the expected reaction is stored in theses tables. These tables will be used during the installation process and will be helpful for the enforcement of the policy. These tables can be deduced from the definition of $\mathbb{P}_{\mathcal{I}\cup\mathcal{X}}$ and $\mathbb{P}_{\mathcal{C}}$.

The four rules given hereafter define the semantics of the BSPL language.

V. Composition of Policies

In this section, we present how to compose two information flow policies designed by different owners of data and containers. We give a formula that permits to compute an information flow policy $\mathbb{P}^f$ resulting from the composition of two policies $\mathbb{P}^1$ and $\mathbb{P}^2$ and we denote it by:

$$\mathbb{P}^f = (\mathbb{P}^f_{\mathcal{C}_f}, \mathbb{P}^f_{\mathcal{I}_f\cup\mathcal{X}_f}, \mathbb{T}^f_{\mathcal{C}_f}, \mathbb{T}^f_{\mathcal{I}_f\cup\mathcal{X}_f}) = \mathbb{P}^1 \oplus \mathbb{P}^2 = (\mathbb{P}^1_{\mathcal{C}_1}, \mathbb{P}^1_{\mathcal{I}_1\cup\mathcal{X}_1}, \mathbb{T}^1_{\mathcal{C}_1}, \mathbb{T}^1_{\mathcal{I}_1\cup\mathcal{X}_1})) \oplus (\mathbb{P}^2_{\mathcal{C}_2}, \mathbb{P}^2_{\mathcal{I}_2\cup\mathcal{X}_2}, \mathbb{T}^2_{\mathcal{C}_2}, \mathbb{T}^2_{\mathcal{I}_2\cup\mathcal{X}_2})$$

The tables $\mathbb{T}_{\mathcal{C}}$ and $\mathbb{T}_{\mathcal{I}\cup\mathcal{X}}$ are used for the behaviour in case of unknown containers/data and to extract the owners of the containers/data and the authors of the policies. We suppose here that data are uniquely identified (through their initial location for instance) and that an identifier only corresponds to a piece of information. We also suppose that users, persistent containers and processes are uniquely identified. The sensitive non-executed pieces of information manipulated by $\mathbb{P}_f$ is denoted by $\mathcal{I}_f$ and is equal to $\mathcal{I}_1 \cup \mathcal{I}_2$. Codes that will later be executed by processes and that are considered as sensitive for the policy $\mathbb{P}_f$ are $\mathcal{X}_1 \cup \mathcal{X}_2$. Consequently $(\mathcal{I}_f \cup \mathcal{X}_f)$ is equal to $((\mathcal{I}_1 \cup \mathcal{I}_2) \cup (\mathcal{X}_1 \cup \mathcal{X}_2))$.

In the same way, containers of information manipulated by $\mathbb{P}_f$ are $\mathcal{C}_f = \mathcal{PC}_f \cup U_f \cup \mathcal{E}xecute(\mathcal{X}_f)$ where users are $\mathcal{U}_f = \mathcal{U}_1 \cup \mathcal{U}_2$, persistent containers are $\mathcal{PC}_f = \mathcal{PC}_1 \cup \mathcal{PC}_2$ and lastly processes $\mathcal{E}xecute(\mathcal{X}_f)$ are $\mathcal{E}xecute(\mathcal{X}_1) \cup \mathcal{E}xecute(\mathcal{X}_2)$.

The final tables are $\mathbb{T}^f_{\mathcal{C}_f} = \mathbb{T}^1_{\mathcal{C}_1} \cup \mathbb{T}^2_{\mathcal{C}_2}$ and $\mathbb{T}^f_{\mathcal{I}_f\cup\mathcal{X}_f} = \mathbb{T}^1_{\mathcal{I}_1\cup\mathcal{X}_1} \cup \mathbb{T}^1_{\mathcal{I}_2\cup\mathcal{X}_2}$.

We define the function information owner $ownerl : \mathcal{I}_f \rightarrow \mathcal{U}$ ($\mathcal{U}$ is the set of users owning containers or data in one of the two policies, we can build it with the field owner of the tables). $ownerl(i) = \mathbb{T}^f_{\mathcal{I}_f\cup\mathcal{X}_f}(i)^3$ (the third field of the table, the owner). We also define the function container owner $ownerc : \mathcal{C}_f \rightarrow \mathcal{U}$. $ownerc(c) = \mathbb{T}^f_{\mathcal{C}_f}(c)^3$. The function $behaviour : \mathcal{I}_f \cup \mathcal{C}_f \rightarrow \{AlwaysAccept; NeverAccept; Ask\}$ is used to access the field behaviour of the tables. We define the function $authors$, which returns the set of the authors of

Rule 1: construction of the data policy ($\mathbb{P}_{\mathcal{I}\cup\mathcal{X}}$)

```
data_identification   = {
        alias = al;
        origin = or;
        owner = own;
        data_type = type;
        reaction = reac;
        case_of_unknown
        _containers = beha; },
[[can_flow   = {
        into = c;
        mixed_with_data = mix; }
 can_flow    = {...     }]]
```

if $(al, \_, \_, \_, \_) \notin \mathbb{T}_{\mathcal{I}\cup\mathcal{X}}$
then $\mathbb{T}_{\mathcal{I}\cup\mathcal{X}} = \mathbb{T}_{\mathcal{I}\cup\mathcal{X}} \cup (al, or, own, type, reac, beha)$;
for each `can_flow` element do
$\mathbb{P}_{\mathcal{I}\cup\mathcal{X}}(al) \leftarrow (c, mix)$
$\forall ix \in mix\ \mathbb{P}_{\mathcal{I}\cup\mathcal{X}}(ix) \leftarrow (c, (mix \cup \{al\}) \setminus \{ix\})$
$\mathbb{P}_{\mathcal{C}}(c) \leftarrow \mathbb{P}_{\mathcal{C}}(c) \cup (mix \cup \{al\})$ done

Rule 2.1: Construction of the Container Policy ($\mathbb{P}_{\mathcal{C}}$): User's Case

```
container_identification   = {
        alias = u;
        origin = "";
        owner = "";
        container_type = User;
        reaction = reac;
        case_of_unknown
        _data = beha; }
can_receive   = {
        mixture_of_data = mix; }
```

if $(u, "", "", User, \_) \notin \mathbb{T}_{\mathcal{C}}$
then $\mathbb{T}_{\mathcal{C}} = \mathbb{T}_{\mathcal{C}} \cup (u, "", "", User, reac, beha)$;
$\mathbb{P}_{\mathcal{C}}(u) \leftarrow mix$;
$\forall ix \in mix,\ \mathbb{P}_{\mathcal{I}\cup\mathcal{X}}(ix) \leftarrow (c, (mix \setminus \{ix\}))$

Rule 2.2: Construction of the Container Policy ($\mathbb{P}_{\mathcal{C}}$): File's Case

```
container_identification   = {
        alias = al;
        origin = or;
        owner = own;
        container_type = File;
        reaction = reac;
        case_of_unknown
        _data = beha; }
can_receive   = {
        mixture_of_data = mix; }
```

if $(al, \_, \_, \_, \_) \notin \mathbb{T}_{\mathcal{C}}$
then $\mathbb{T}_{\mathcal{C}} = \mathbb{T}_{\mathcal{C}} \cup (al, or, own, reac, beha)$;
$\mathbb{P}_{\mathcal{C}}(al) \leftarrow mix$
$\forall ix \in mix,\ \mathbb{P}_{\mathcal{I}\cup\mathcal{X}}(ix) \leftarrow (c, (mix \setminus \{ix\}))$

Rule 2.3: Construction of the Container Policy ($\mathbb{P}_{\mathcal{C}}$): Process Case

```
container_identification   = {
        alias = al;
        origin = or;
        owner = own;
        container_type = Process;
        reaction = reac;
        case_of_unknown
        _data = beha; }
can_receive   = {
        mixture_of_data = mix;  }
```

if $(al, \_, \_, \_, \_) \notin \mathbb{T}_{\mathcal{C}}$
then $\mathbb{T}_{\mathcal{C}} = \mathbb{T}_{\mathcal{C}} \cup (\mathcal{E}xecute(al), or, own, reac, beha)$;
$\mathbb{P}_{\mathcal{C}}(\mathcal{E}xecute(al)) \leftarrow mix$
$\forall ix \in mix,\ \mathbb{P}_{\mathcal{I}\cup\mathcal{X}}(ix) \leftarrow (c, (mix \setminus \{ix\}))$

Fig. 1: Semantics rules of BSPL

a policy (the set of owners of data/containers appearing in the table of this policy). We finally use $request : \mathcal{U} \times (\mathcal{I}_f \cup \mathcal{C}_f) \times \mathscr{P}(\mathcal{I}_f \cup \mathcal{X}_f) \rightarrow \{True; False\}$, boolean function as the answer from the user to the request of a policy.

The building function $\mathbb{P}^f_{\mathcal{C}_f}$ is formally defined by the definition 2. Intuitively this definition states that the sets I associated with c are either in the policy $\mathbb{P}^1$ (resp. $\mathbb{P}^2$) with every data and c owned by an author of $\mathbb{P}^1$ (resp. $\mathbb{P}^2$) or correspond to an intersection of the two policies on the containers for which both policies specify mixes of information. The rest of the sets in the definition are used to collect the cases in which the behaviour specified for an unknown container or an unknown data by an owner allows the result policy to contain an element.

Definition 2 explains the construction of $\mathbb{P}^f_{\mathcal{I}_f \cup \mathcal{X}_f}$. This definition is dual (thanks to the coherence property) with the definition of $\mathbb{P}^f_{\mathcal{C}_f}$.

With this definition, we are able to prove that the policy computed by composition of two consistent policies is still a consistent policy. The proof is not detailed here due to the limitation of space.

Definition 2 (composition of policies):
$\mathbb{P}^1$ and $\mathbb{P}^2$ two information flow policies. The policy $\mathbb{P}^f = (\mathbb{P}^f_{\mathcal{C}_f}, \mathbb{P}^f_{\mathcal{I}_f \cup \mathcal{X}_f})$ obtained by composition of $\mathbb{P}^1$ and $\mathbb{P}^2$ is denoted by $\mathbb{P}^1 \oplus \mathbb{P}^2$ and is defined by $\mathbb{P}^f_{\mathcal{C}_f}$ and $\mathbb{P}^f_{\mathcal{I}_f \cup \mathcal{X}_f}$ where : $(\mathcal{I}_f \cup \mathcal{X}_f) = ((\mathcal{I}_1 \cup \mathcal{I}_2) \cup (\mathcal{X}_1 \cup \mathcal{X}_2))$, and $\mathcal{C}_f = \mathcal{C}_1 \cup \mathcal{C}_2$

$$
\begin{aligned}
\mathbb{P}^f_{\mathcal{C}_f}(c) = \quad & \\
& \{I \in \mathbb{P}^1_{\mathcal{C}_1}(c) | \forall i \in I, \\
& owner l(i) \in authors(\mathbb{P}^1) \wedge (ownerc(c) \in authors(\mathbb{P}^1))\} \\
\cup \; & \{I \in \mathbb{P}^2_{\mathcal{C}_2}(c) | \forall i \in I, \\
& owner l(i) \in authors(\mathbb{P}^2) \wedge (ownerc(c) \in authors(\mathbb{P}^2))\} \\
\cup \; & \{(I_1 \cap I_2) | I_1 \in \mathbb{P}^1_{\mathcal{C}_1}(c) \\
& \wedge I_2 \in \mathbb{P}^2_{\mathcal{C}_2}(c)\} \\
\cup \; & \{I \in \mathbb{P}^1_{\mathcal{C}_1}(c) | (ownerc(c) \in author(\mathbb{P}^2)) \\
& \wedge (behaviour(c) = AlwaysAccept \\
& \vee (behaviour(c) = Ask \\
& \wedge request(ownerc(c), c, I))))\} \\
\cup \; & \{I \in \mathbb{P}^2_{\mathcal{C}_2}(c) | (ownerc(c) \in author(\mathbb{P}^1)) \\
& \wedge (behaviour(c) = AlwaysAccept \\
& \vee (behaviour(c) = Ask \\
& \wedge request(ownerc(c), c, I \cup \{ix\})))\} \\
\cup \; & \{I \in \mathbb{P}^1_{\mathcal{C}_1}(c) | \exists ix \in I, owner l(ix) \in authors(\mathbb{P}^2) \\
& \wedge (ownerc(c) \in author(\mathbb{P}^1)) \\
& \wedge (behaviour(ix) = AlwaysAccept \\
& \vee (behaviour(ix) = Ask \\
& \wedge request(ownerl(ix), ix, I \cup \{ix\})))\} \\
\cup \; & \{I \in \mathbb{P}^2_{\mathcal{C}_2}(c) | \exists ix \in I, owner l(ix) \in authors(\mathbb{P}^1) \\
& \wedge (ownerc(c) \in author(\mathbb{P}^2)) \\
& \wedge (behaviour(ix) = AlwaysAccept \\
& \vee (behaviour(ix) = Ask \\
& \wedge request(ownerl(ix), ix, I \cup \{ix\})))\}
\end{aligned}
$$

$$
\begin{aligned}
\mathbb{P}^f_{\mathcal{I}_f \cup \mathcal{X}_f}(ix) = \quad & \\
& \{(c, I) \in \mathbb{P}^1_{\mathcal{I}_1 \cup \mathcal{X}_1}(ix) | \forall i \in (I \cup \{ix\}), \\
& owner l(i) \in authors(\mathbb{P}^1) \wedge (ownerc(c) \in authors(\mathbb{P}^1)) \\
\cup \; & \{(c, I) \in \mathbb{P}^2_{\mathcal{I}_2 \cup \mathcal{X}_2}(ix) | \forall i \in (I \cup \{ix\}), \\
& owner l(i) \in authors(\mathbb{P}^2) \wedge (ownerc(c) \in authors(\mathbb{P}^2)) \\
\cup \; & \{(c, I_1 \cap I_2) | (c, I_1) \in \mathbb{P}^1_{\mathcal{I}_1 \cup \mathcal{X}_1}(ix) \\
& \wedge (c, I_2) \in \mathbb{P}^2_{\mathcal{I}_2 \cup \mathcal{X}_2}(ix)\} \\
\cup \; & \{(c, I) \in \mathbb{P}^1_{\mathcal{I}_1 \cup \mathcal{X}_1}(ix) | (ownerc(c) \in author(\mathbb{P}^2)) \\
& \wedge (behaviour(c) = AlwaysAccept \\
& \vee (behaviour(c) = Ask \\
& \wedge request(ownerc(c), c, I \cup \{ix\})))\} \\
\cup \; & \{(c, I) \in \mathbb{P}^2_{\mathcal{I}_2 \cup \mathcal{X}_2}(ix) | (ownerc(c) \in author(\mathbb{P}^1)) \\
& \wedge (behaviour(c) = AlwaysAccept \\
& \vee (behaviour(c) = Ask \\
& \wedge request(ownerc(c), c, I \cup \{ix\})))\} \\
\cup \; & \{(c, I) \in \mathbb{P}^1_{\mathcal{I}_1 \cup \mathcal{X}_1}(ix) | owner l(ix) \in authors(\mathbb{P}^2) \\
& \wedge (ownerc(c) \in author(\mathbb{P}^1)) \\
& \wedge (behaviour(ix) = AlwaysAccept \\
& \vee (behaviour(ix) = Ask \\
& \wedge request(ownerl(ix), ix, I \cup \{ix\})))\} \\
\cup \; & \{(c, I) \in \mathbb{P}^2_{\mathcal{I}_2 \cup \mathcal{X}_2}(ix) | owner l(ix) \in authors(\mathbb{P}^1) \\
& \wedge (ownerc(c) \in author(\mathbb{P}^2)) \\
& \wedge (behaviour(ix) = AlwaysAccept \\
& \vee (behaviour(ix) = Ask \\
& \wedge request(ownerl(ix), ix, I \cup \{ix\})))\}
\end{aligned}
$$

VI. USING BSPL: EXAMPLES

A. Example of policy in BSPL

We propose in this section a part of what could be a flow policy of a game application for Android, e.g., AngryBirdsSpace. We have studied this application and found that the pieces of information linked to the application (as its code, its own data) can flow to 61 containers of information (more precisely 22 processes and 39 files). Our study has also shown that once launched the game receives piece of information from more than 30 differents origins.

If we were developers of this application, we would have to specify a BSPL policy that takes into account this behavior. We present in Fig. 2 a part of this policy. This part defines a sensitive piece of information, which is the content of the application package file `/data/app/com.rovio.angrybirds-space.ads-1.apk` aliased as *AngryBirdSpace*. This piece of information is owned by an owner defined by the Android operating system during the installation process. It is not an executed information (i.e., it is of type I). In case of detection of an illegal flow, the monitor raises an alert but does not forbid the flow. The default behavior is used in case of an unknown container. The rest of the `<Data-policy>` defines the containers this piece of information can flow into. More precisely it specifies that this application may create a file where it will store its cookies, will create a file named `highscores.lua.tmp` that will contain the scores, will use the vibrate facility and will ask the music player to play its own music. This application will create at least one container of information (a process) named `ybirdsspace.ads` that will receive data, in particular it will receive the piece of information identified just above *AngryBirdSpace*.

 ISBN: 978-1-61208-298-1

```
<BSPL-policy>
  <Data-policy>
    <data>
      <data_identification
      alias ="AngryBirdSpace"
      origin="/data/app/com.rovio.angrybirdsspace
               .ads-1.apk"
      owner="App_2"
      type="I"
      reaction="Alert"
      case_of_unknown_containers="Ask"
      />

      <can_flow
          into="/data/data/com.rovio.angrybirdsspace
                 .ads/databases/cookiedb-journal"
          mixed_with_data=""
        />
        <can_flow
          into="/data/data/com.rovio.angrybirdsspace
                 .ads/files/highscores.lua.tmp"
          mixed_with_data=""
        />
        <can_flow
          into=""
          mixed_with_data="/sys/devices/virtual/
                           timed_output/vibrator
                           /enable"
        />
        <can_flow
          into="android.musicfx"
          mixed_with_data=""
        />
      <can_flow>
          into="system_server"
          mixed_with_data=""
        />
        <can_flow
          into="system_server"
          mixed_with_data=""
        />
    </data>
  </Data-policy>

  <Containers-policy>
    <container>
      <container_identification
        alias=" ybirdsspace.ads"
        origin=" ybirdsspace.ads"
        owner="app_2"
        container_type="Process"
        reaction="Alert"
        case_of_unknown_data="Ask"
      />
      <can_receive
          mixture_of_data="AngryBirdSpace"
        />
      </can_receive>
    </container>
  </Containers-policy>

</BSPL-policy>
```

Fig. 2: A part of a BSPL policy for Angry Bird Space

```
<BSPL-policy>
  <Data-policy>
    <data>
      <data_identification
        alias ="file"
        origin="/data/local/tmp/file"
        owner="me"
        type="I"
        reaction="Alert"
        case_of_unknown_containers="Ask"
      />
      <can_flow>
          into="md5sum"
          mixed_with_data=""
        />
        <can_flow
          into="hash"
          mixed_with_data=""
        />
    </data>
  </Data-policy>
<Containers-policy>
    <container>
      <container_identification
        alias="hash"
        origin="/data/local/tmp/hash"
        owner="root"
        container_type="File"
        reaction="Alert"
        case_of_unknown_data="NeverAccept"
      />
    </container>
    <container>
      <container_identification
        alias="md5sum"
        origin="/system/bin/md5sum"
        owner="me"
        container_type="Process"
        reaction="Alert"
        case_of_unknown_data="AlwaysAccept"
      />
    </container>
  </Containers-policy>
</BSPL-policy>
```

Fig. 3: Hashing Example: Initial Policy

B. Example of Composition

In this subsection, we describe another example to illustrate a simple case of composition. We propose a scenario involving a hashing function. Notice that in this example the owner of all data and all containers is root, but it could be anything else and in particular the two processes could be owned by different users. The hashing function `md5sum` is installed in our system. We have also a file named FILE containing sensitive values ; we want to forbid the reading of these values. However, we trust the application `md5sum` and we permit to compute and publish the hash value of FILE using `md5sum` in HASH. Fig. 3 gives this simple policy.

With such a policy, using the command `md5sum /data/local/tmp/file > /data/local/tmp/hash` does not raise an alert while

 ISBN: 978-1-61208-298-1

```
<Containers-policy>
  <container>
    <container_identification
      alias="sha1sum"
      origin="/system/bin/sha1sum"
      owner="root"
      container_type="Process"
      reaction="Alert"
      case_of_unknown_data="NeverAccept"
    />
    <can_receive
        mixture_of_data="file"
      />
  </container>
</Containers-policy>
```

Fig. 4: Hashing Example: sha1sum Policy

the command `sha1sum /data/local/tmp/file > /data/local/tmp/hash` does. Indeed, the policy does not take `sha1sum` into account at this point. Notice that the behavior "`ask`" is associated to unknown containers.

Fig. 4 illustrates: `md5sum` is now considered unsafe and the owner of `sha1sum` wants to replace `md5sum` by `sha1sum`. This policy expresses that the owner of `sha1sum` requests the right for the processes running `sha1sum` to contain *file*.

We can now compose these two last policies. In our scenario, this composition requires the intervention of the owner of *file* since the behavior specified in the case of an unknown container is "`ask`". If this behavior was "`alwaysaccept`" or "`neveraccept`", the composition could be realized automatically. The owner of *file* accepts since he wants to replace `md5sum` by `sha1sum`. Fig. 5 shows the result of the composition.

In this composed policy, both `md5sum` and `sha1sum` have the right to contain *file* and `hash` can still contain *file* too. There are three containers and all of them can contain *file*. Now the command `sha1sum /data/local /tmp/file > /data/ local/tmp/hash` no longer raises an alert. However, the command `md5sum /data/local/tmp/file > /data/local/ tmp/hash` neither raises an alert. The final step is thus to remove the right of `md5sum` to contain *file*. This step can only be realized by a manual intervention from the owner of `md5sum`. The final resulting policy is almost the same as in Fig. 5, but `md5sum` can no longer contain *file*. After this step, the command `md5sum /data/local/tmp/file > /data/local/tmp/hash` raises an alert. The lines of the policy that disappear are all the lines concerning `md5sum`.

This scenario illustrates a simple use of BSPL, but it shows the expressiveness of the language. Each owner can express a policy for its data and containers and let the composition mechanism merge the policies to produce the policy to be finally enforced by Blare on the system. This example also shows that it is possible to restrain access to some pieces of information to specific processes executing a code considered as safe.

```
<?xml version="1.0" encoding="UTF-8"?>
<BSPL-policy>
  <Data-policy>
    <data>
      <data_identification
      alias ="file"
      origin="/data/local/tmp/file"
      owner="root"
      type="I"
      reaction="Alert"
      case_of_unknown_container="Ask"
      />
      <can_flow
        into="md5sum"
        mixed_with_data=""
      />
      <can_flow
        into="sha1sum"
        mixed_with_data=""
       />
       <can_flow
        into="hash"
        mixed_with_data=""
        />
    </data>
  </Data-policy>
  <Containers-policy>
    <container>
      <container_identification
      alias="hash"
      origin="/data/local/tmp/hash"
      owner="root"
      container_type="File"
      reaction="Alert"
      case_of_unknown_data="NeverAccept"
      />
    </container>
    <container>
      <container_identification
      alias="md5sum"
      origin="/system/bin/md5sum"
      owner="root"
      container_type="Process"
      reaction="Alert"
      case_of_unknown_data="AlwaysAccept"
      />
    </container>
    <container>
      <container_identification
      alias="sha1sum"
      origin="/system/bin/sha1sum"
      owner="root"
      container_type="Process"
      reaction="Alert"
      case_of_unknown_data="NeverAccept"
      />
    </container>
  </Containers-policy>
</BSPL-policy>
```

Fig. 5: Hashing Example: Composed Policy

VII. Conclusion and future work

In this paper, we have presented BSPL, a language designed to specify an information flow policy for developpers of

applications of users. This language is dedicated to the `Blare` monitor.

The information flow policies expressed using BSPL specify which and how sensitive pieces of information may be combined and in which containers of information these data mixtures can flow.

The language has been presented here and we have proposed precise semantics based on a solid model of information flow published earlier.

We have defined a consistency property for a policy specified in this model and have shown how to compose two consistent policies. Our definition of composition respects the consistency property: a composition of two consistent policies leads to a third consistent policy.

We already have a tool allowing to compute blare tags using a BSPL policy. Thus, we are able to monitor flows spawned by applications running over Android with respect to a BSPL policy.

For our future work, we plan to focus on adding the possibility for users to declassify sensitive data in certain contexts and to administrate their policy.

REFERENCES

[1] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in histar," in *OSDI*, 2006, pp. 263–278.

[2] V. V. T. Tong, A. J. Clark, and L. Mé, "Specifying and enforcing a fine-grained information flow policy : model and experiments," vol. 1, no. 1, 2010, pp. 56–71, this paper was part of the 2nd International Workshop on Managing Insider Security Threats (MIST 2010); Morika, Iwate, Japan (14-15 June 2010).

[3] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[4] P. Efstathopoulos, M. N. Krohn, S. Vandebogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, M. F. Kaashoek, and R. Morris, "Labels and event processes in the asbestos operating system," in *SOSP*, 2005, pp. 17–30.

[5] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *OSDI*, 2010, pp. 393–407.

[6] P. Efstathopoulos and E. Kohler, "Manageable fine-grained information flow," in *EuroSys*, 2008, pp. 301–313.

[7] S. Geller, C. Hauser, F. Tronel, and V. V. T. Tong, "Information flow control for intrusion detection derived from mac policy," in *ICC*, 2011, pp. 1–6.

[8] G. Hiet, V. V. T. Tong, L. Mé, and B. Morin, "Policy-based intrusion detection in web applications by monitoring java information flows," pp. 53–60, 2008.

[9] "Designing information flow policies for android's operating system," in *ICC*, 2012, pp. 976–981.

[10] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. N. Krohn, C. Frey, D. Ziegler, M. F. Kaashoek, R. Morris, and D. Mazières, "Labels and event processes in the asbestos operating system," *ACM Trans. Comput. Syst.*, vol. 25, no. 4, 2007.

[11] S. Geller, "Information flow and execution policy for a model of detection without false negatives," in *Network and Information Systems Security (SAR-SSI), 2011 Conference on*, 2011, pp. 1–9.

 ISBN: 978-1-61208-298-1