Extend Python Information Flow Analyzer in Distributed Systems

Xuran Cai, Jayden Ye

August 25, 2022

1 Introduction

In recent years, distributed systems have become more and more popular in many areas. With more and more complex distributed designs coming into the world, the information flow becomes hard to track, which means some secure information may be leaked when flowing to unwanted areas. Hence, preventing secure information from flowing into insecure hosts becomes a considerable problem. [5] The mainstream way to solve this problem is using hardware to provide memory encryption and hence doing all the computations which need secure information in a safe area. However, such hardware like Intel sgx and AMD trust zone is always expensive and cannot take care of the whole system. If the secure information flows out of the enclave created by the hardware by accident, such mechanisms cannot track that piece of information anymore. After realizing that, we tried to find some alternative way to track all the information flow within the whole complex distributed system.

Our solution is to find a language-based tool called the information flow control system (IFC). The information flow control system is an extension of current languages, which binds labels for each piece of information within a program to show their security level. These labels would be checked when one piece of information wants to flow to another. There are many existing IFC with different label algorithms, and some of them already have distributed versions called decentralized information flow control systems (DIFC). However, none of these IFCs could satisfy the complex situation of distributed systems. For example, the most popular IFC, JIF, and related DIFC Fabric

system mainly use a static label algorithm that generates all the labels before computation. The static label is enough for a local host, but it is too restrictive and may reject some safe flow and may leak information about loop statements in complex distributed systems. Because of this, we wanted a better algorithm for distributed systems. We gained some ideas from pifthon, which uses a semi-dynamic labeling algorithm and try to combine both dynamic and static labels with a monotonic updated program counter. With all these new ideas, this a special label algorithm could solve most tricky problems from other IFCs.

Hence, after learning all these ideas, we created a new distributed information flow system. The system consists of two major layers, the store node, and the worker node. These nodes follow the policy of dynamic labeling strategy with a monotonic pc.

Our main contributions includes:

1. Design and implement a new policy and labeling algorithm based on the existing modern information flow controller Pifthon, to make it work on distributed flow system built with Python.

2. Design the architecture of the distributed system. Implement store nodes that carry the data and execution code and worker nodes that do the validation and calculation.

3. Validate our design on a Friend Map application as a case study. And explain how our system can be extended to other scenarios with security constraints.

2 Label Mechanism

All the IFCs are based on label mechanics introduced by Denning's security lattice mode[1]. In that model, a program should be considered as $< N, P, SC, \bigoplus, \rightarrow >$, where N denotes the set of all the information (variables, objects, and maybe files), P denotes the set of all the processes, in other words, all the information flows in the program. In SC, it contains different security policies, which are called labels. Each element in N should

correspond to an element in SC, and we could use a function L to map information to its related label. $\forall x \in N, \exists \underline{x} \in SC, L(x) = \underline{x}$.

The operation $\bigoplus$ called "join" is a binary operation to combine labels in SC, which could find and return the overlap area of two labels. It is used when combining two pieces of information, for example, $L(x+y) = \underline{x+y} = \underline{x} \bigoplus \underline{y}$. The relation $\rightarrow$ called "can-flow relation" is defined on a pair of labels. We state $\underline{x} \rightarrow \underline{y}$ if and only if the information in x can flow to y(information associated with x can affect the information associated with y). Only if all the flows in P are among the $\rightarrow$ relation, could we state that this program is safe.

In JIF, all the labels are static, which means all the labels are generated before computation and would not change during computation. Here is an example: in the program

```
while(a):
    b = c
d = e
```

There are two information flows and five variables in the short piece of program. For the first information flow (b=c), as it is inside the while loop, both a and c may affect the information in b, so we could generate the label constrain like $\underline{a} \bigoplus \underline{c} \rightarrow \underline{b}$. Similarly, for the second one, we could generate label constrain $\underline{e} \rightarrow \underline{d}$. As long as labels satisfy the two constraints, JIF would consider this piece program secure. However, if we found e is changed, we could know that a becomes false, which means information in a may be leaked by accessing to e. Hence we need to find a way for this situation.

In pifthon, it adds another label called program counter(pc) to keep tracking the track the least upper bound of all the security levels till now, and it should be able to flow to all the destination of any information flows, which means it is dynamic and would keep updating during computation. In this example, at first, pc is public and can flow into anywhere. After the first information flow, we should update pc as $\underline{a} \bigoplus \underline{c}$. Hence when the loop ends and comes to the second information flow, we need first check if $pc \rightarrow \underline{d}$. Hence no information can be leaked due to the loop statement.

Another problem with JIF static label algorithm is that sometimes it may reject safe flow. For example, in the following program

```
a = b
c = a
a = d
e = a
```

As both b and d have flowed to a at some time, so we would generate $\underline{b} \bigoplus \underline{d} \rightarrow \underline{a}$. As a flow to c, $\underline{a} \rightarrow \underline{c}$, which means $\underline{b} \bigoplus \underline{d} \rightarrow \underline{c}$ according to JIF. However, if we go through the program, we would find that d will not affect the information in c, which means even if d cannot flow to c, this program can be safe, but in that case, JIF would consider it as illegal flow due to the static label algorithm.

To solve this, pifthon adds some dynamic labels, which means instead of only generating constraints, the label would be updated during computation just like pc.In this example, if we first set a as public, then after b flow to a, $\underline{a} = \underline{b}$, and hence, when a flow to c, we only need to check if b can flow to c without considering d. Only after d flow to a, the label for a updates to $\underline{b} \bigoplus \underline{d}$, the following flow e=a need to check if $\underline{b} \bigoplus \underline{d} \rightarrow \underline{e}$.

With the two ideas learned from pifthon, we begin to design our distributed system.

3 System Architecture

In our design, we borrow the idea of store node, and worker node in Fabric [3]. Store nodes store objects persistently. They are able to provide and update data when requested. Worker nodes perform computation after getting code and data copies from store nodes. These nodes can interact and share information without prior arrangement. There is no centralized control over admission: new nodes, even untrustworthy nodes, can join the system freely. [3] The major difference between our system and Fabric is that Fabric works on Java, and we implement this system in Python, and we leave out

the unnecessary dissemination nodes that aim at lower latency. Thus our system has two layers instead of three. And to make sure things work well with the information flow security, Fabric uses JIF [4], and we utilize ideas introduced in Pifthon [2]. Our system manages to use a dynamic labeling strategy with a monotonic program counter (pc), which is more flexible and provides mutable labels.

3.1 Store Node

Naming For security purposes in the distributed system, every node should have both ID and its corresponding key. Only the creators themselves own the key, and others can only have the public ID. We use the location of the configuration JSON file and creation time as the unique key. To represent that node in the distributed system, we encrypted it with the modern SHA256 hashmap algorithm to get the ID. For example, key "D:/Downloads/CS 704/project/Distributed-Flow-Controller-Python/alice.json_21:20:33" is encrypted as ID "00295eabeceb3182c5e2fe80ea4f9f3f8a0729481748f399eaeb15ecfb1862fb". All other nodes can have the public ID, but the only way to pass authentication is for the requester provides the key that generated the same public ID.

Labels We decide to use RWFM labels in our system. They can be represented as (o, R, W), where o denotes the owner of the information and policy, R, the readers, denotes the set of subjects allowed to read the information, and W, the writers, refers to the set of subjects who have influenced the information so far. [2] It might be worth mentioning that the writer here has a different meaning from the one in file system permission control. Initially, their reading and writing list contain only themselves when initialized. To provide more features in the system, someone can allow access to all nodes at first if configured in the JSON file. With some interaction between nodes, the labels can change. Requesters can only get the data with an unencrypted key if they are in the reading list of the store node.

Data Every store node contains JSON-format data along with its RWFM labels. All nodes contain their own information in the 'data' property and security information in the 'label' property. The data in our following case study is information including name, address, and friend

list. The store node will send back all the data to authenticated requests. They are also able to modify the data if update requests are made. Some store nodes may contain execution codes and intermediate results generated by these execution codes. These nodes are closely related to the working process of worker nodes.

3.2 Worker Node

To start a worker node, the creator must send their secret key and the corresponding store node ID that stores the functional code to execute. The worker node might need extra data from other store nodes when executing them. Sometimes it might need data from results generated previously. All these reading operations is a reserved function called `getData()`. With an ID address, the system will start to request the corresponding store node to fetch data. Of course, this operation needs to check if the worker node creator ID is in the reader list of that requested data store node. Similarly, the system has another write function `setData()`. Only the owner is allowed to modify the data inside the store node. For intermediate results, they are initialized and updated in newly created store nodes.

3.3 DNS

To make sure nodes can successfully and correctly communicate with each other, a Domain Name Server is vital. Because all the store nodes display themselves with encrypted ID, it would be necessary to map the ID to its actual web address so that others can get their way to them. Once a node is created, it should notify the DNS and log its IP and portal so future requests can be smoothly processed.

3.4 Labeling Strategy

RWFM label label with structure (o, R, W)

join operation($\oplus$) for labels $\underline{x} = (-, R_x, W_x)$ and $\underline{y} = (-, R_y, W_y)$, $\underline{x} \oplus \underline{y} = (-, R_x \bigcap R_y, W_x \bigcap W_y)$

can-flow relation($\rightarrow$) for labels $\underline{x} = (-, R_x, W_x)$ and $\underline{y} = (-, R_y, W_y)$, $\frac{R_x \supseteq R_y,\ W_x \subseteq W_y}{\underline{x} \rightarrow \underline{y}}$

A typical RWFM contains three parts, owner, readers, and writers, which would be stated as (owner,[],[]). During initialization, the variables with static labels would be provided by the user and stay constant during analysis. Also, new variables would be created with a label (S,[*],[]) which means anyone could read it, and nobody has influenced it yet. This label would work as a dynamic label and change during analysis when we define it as the "join" of two labels. In this model, "join"($\oplus$) would generate a new label with the intersection of the two reader sets and the union of two writer sets. For example, if variable a with label (S,[a,b,c],[a]) and b with label (S, [b,c,d],[b]). If we create new variable c as the combination of a and b (c=a+b), the label generated for c would be (S,[b,c],[a,b]).

Another important definition is about "can-flow relation"($\rightarrow$). In this model,$\underline{a} \rightarrow \underline{b}$ if and only if the read set of b is a subset of a and the writer set of a is a subset of b.

4 Case Study

We will use friend map as a case study to show how our distributed system with information flow control works. This friend map case has clear enough trust relationships and communication patterns, which is suitable to demonstrate the expressiveness of our system.

In a social network, users are allowed to create a map displaying all their friends' locations. In 1, we have a user Alice, and her friends John and Bob in a simple network. Alice intends to put all of their friends on a world map and display the map to her friends. Alice herself, of course, can see the generated map, since all people on that map is Alice's friend. Ideally, everyone on this map should see Alice's friend map because all of them are Alice's friends. However, one of her friends, Bob, does not want anyone other than his friends to see his location. Since John is not Bob's friend, John cannot see Alice's friend map because Bob is on it, even though he is one of Alice's friends. We will show how our system handles this difficulty in the following paragraphs.

We first start the DNS and create three store nodes, Alice, Bob, and John, based on a given predefined JSON file. The initial status of these nodes should have the following RWFM. Note that A, B, and J stand for Alice, Bob, and John. And these IDs are encrypted when executing. We use

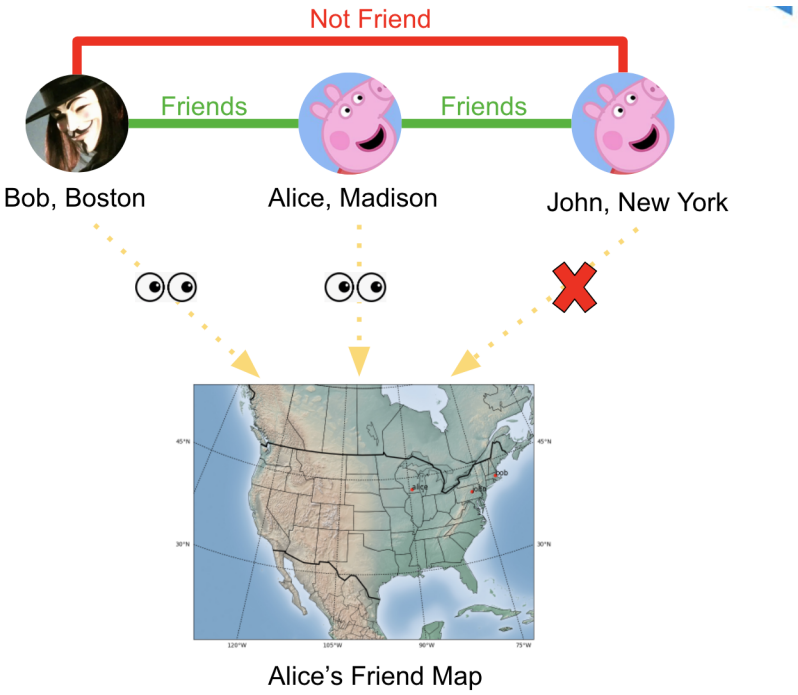

Figure 1: Friend Map Example

one letter to represent subjects for simplicity and clearness here.

```
{ID: A, O: A, R:[A, B, J], W:[A]}
{ID: B, O: B, R:[A, B], W:[B]}
{ID: J, O: J, R:[A, B, J], W:[J]}
```

In the starting status (also called global variable in Pifthon), they only include themselves and their trusted friend ID in their reader label. The exception is that John allows reading access to all nodes in its configuration. For this initialization, we see that Alice allows her two friends, Bob and John, to read her data, and Bob only allows her friend Alice to read its location. But John does not mind and allows everyone to get his information.

We create another store node that stores the `get_friends_location` function executable code and give it to the worker node once requested. The code is tailored for our system. getData and setData are reserved functions so

that our Python parser can use that to communicate with other store nodes. Other than that, the code is straightforward, just a for loop to get all her friends' address information.

```
result = newNode([])
for id_ in getData(id)['friends']:
    setData(result, result+[getData(id_)['address']])
```

In this `get_friends_location` function, it fetches all the friend ID information from Alice store node. And then, it recursively gets address information from nodes with these IDs. Because the caller of this worker node is Alice, who has access to all the friends, the problem can run successfully. And it will store the result in another newly created store node whose final label is

```
{ID: A_map, O:A, R:[A, B], W:[A, B, J]}
```

The labeling generation process can be explained step to step. Initially, the program counter and the result label are empty.

```
A_map: {R:[*], W:[]}
pc: {R:[*], W:[]}
```

In the first loop, it tries to fetch Alice's information and do the label checking. If that passes, it will update the program counter. Because there is an assignment operation to the result, the label of the result needs to be updated to the pc.

```
alice+result -> result
pc = alice+result
# pc: {R: [A, B, J], W: [A]}
result = pc
# result: {R: [A, B, J], W: [A]}
```

In the second loop, another of her friends, bob, is added to pc, which updates the reading and writing list as well. The reader list takes intersection, and the write list takes union operation.

```
bob+alice+result -> result
pc = bob+alice+result
# pc: {R: [A, B], W: [A, B]}
result = pc
# result: {R: [A, B], W: [A, B]}
```

So as the third loop. The pc here is monotonic, and the final label of the result is equivalent to alice⊕bob⊕john.

```
john+bob+alice+result -> result
pc = john+bob+alice+result
# pc: {R: [A, B], W: [A, B, J]}
result = pc
# result: {R: [A, B], W: [A, B, J]}
```

As a result, when John wants to access the result node, he will be denied because he does not have permission. However, if Bob wants to access the generated node, he should be granted.

5 Conclusion

In this work, we have compared different labeling strategies in this field and designed a distributed information flow control system based on Python with a dynamic labeling strategy. The system has achieved decentralized connections. And as a distributed system, it has a degree of cryptography protocols to ensure no unwanted behaviors. We have also demonstrated our system's power using a Friend Map application. The case study should illustrate how our design helps build secure distributed systems and present the behavior of the distributed control flow. And we believe our system can be easily adapted in other security-critical scenarios given corresponding input.

There are some potential improvements to our system. First, we use exclusively dynamic labeling with a monotonic program counter. This strategy can be pretty restrictive. Pifthon uses a semi-dynamic labeling strategy that involves both static and dynamic labeling. We think it might suit a broader range of programs. However, we failed to embed it into our distributed system. Secondly, unlike JIF, our system still needs some reserved functions. Thus only tailored codes can execute. The third issue is usability. We only test things on a local machine. It would make more sense if running on some remote devices as a distributed system. And we might also want to equip data transportation with SSL. Scalability is the last concern. In Fabric, they use another layer called dissemination. These nodes function as caches for better time performance. Although we have not tested things with enormous throughput, we are not expecting great quantitative responsive numbers on our system.

References

[1] Dorothy E Denning. A lattice model of secure information flow. *CACM*, 5:236–243, 1976.

[2] Sandip Ghosal and RK Shyamasundar. Pifthon: A compile-time information flow analyzer for an imperative language. *arXiv preprint arXiv:2103.06039*, 2021.

[3] Jed Liu, Michael D George, Krishnaprasad Vikram, Xin Qi, Lucas Waye, and Andrew C Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 321–334, 2009.

[4] Kyle Pullicino. Jif: language-based information-flow security in java. *arXiv preprint arXiv:1412.8639*, 2014.

[5] Krishnaprasad Vikram. Building distributed systems with information flow control. 2015.