PIF: Python Information Flow

CS254: Formal Methods for Computer Security, Spring 2024

MARIO FARES, VADIM ZARIPOV, ACER IVERSON, and SAM DEPAOLO

1 INTRODUCTION

Information control flow is an effective method to build secure systems and protocols. It allows the framing of information on high and low levels that can be described via security conditions such as noninterference, integrity, and authority.

Modeling this information flow in code is difficult to achieve in most existing languages. Traditional languages such as Python, Java, and C++/C lack careful consideration of these security measures; modeling information flow and limiting this flow from a high-security entity to one of low-security requires extensive additional considerations that present ample opportunity for oversights.

To resolve this shortcoming, security-typed programming languages such as JIF and Jeeves have implemented systems to easily model different protocols with respect to information flow. The goal of our project is to create an additional security-typed programming language in Python. This language will allow for modeling of protocols while respecting information flow to ensure enforcement of security. The language will support integer and float types; support for additional types such as strings and lists will be left to future work.

2 RELATED WORK

There are many other security-typed program languages that have been created to support information flow, with one of the most notable ones being JIF. JIF is an extended version of Java that supports information flow control and access control that are implemented for compile and run time. After tracking information flow and ensuring proper usage, Jif compiles the program to an executable that can be run with a standard Java compiler. Jif acheives this modeling by relying on the addition of labels on objects: these specify the allowed forms of information flow and interaction between objects to allow for automatic verification at compile and runtime [1].

Jeeves is another security-typed language implemented on top of Python. The language respects information flow by allowing the programmer to specify facets, or multiple views, of sensitive values. Each of these facets is in turn specified with a label that determines which facet should be used in the given security context, whether that be a high context or a low one. The programmer details specific policies with labels that are used in runtime to determine values. This thus enables policy-agnostic programs to be written in which sensitive values are used as normal values depending on the labeling while the runtime is relied on for the correct output [2].

3 THE PIF LANGUAGE

3.1 Syntax & Grammar

Type	$\tau := \ num \mid int \mid str \mid bool$
Numeric Operator	$\oplus := \ + \mid - \mid / \mid // \mid * \mid ** \mid \% \mid \& \mid \vert \mid \hat{} \mid << \mid >>$
Comparison Operator	$\bowtie := \ < \mid \leq \mid > \mid \geq \mid == \mid \neq$
Boolean Operator	$\diamond := \ \& \mid \vert \mid \hat{}$
Security Level	$\ell := \ \bot \mid \ldots$
Security Context	$\Gamma := \ \cdot \mid \Gamma, x \mapsto \tau_\ell$
Program Counter	$pc := \ \ell$

3.2 Semantics

Expressions.

$$\textsc{Var}\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \textsc{Num}\ \frac{}{\Gamma \vdash n : num_\bot} \qquad \textsc{Str}\ \frac{}{\Gamma \vdash s : str_\bot} \qquad \textsc{Bool}\ \frac{}{\Gamma \vdash b : bool_\bot}$$

$$\textsc{Comp}\ \frac{\Gamma \vdash x : \tau_{\ell_x} \qquad \Gamma \vdash y : \tau_{\ell_y}}{\Gamma \vdash x \bowtie y : bool_\ell}\ \ell = \ell_x \sqcup \ell_y$$

$$\textsc{NumArith}\ \frac{\Gamma \vdash x : num_{\ell_x} \qquad \Gamma \vdash y : num_{\ell_x}}{\Gamma \vdash x \oplus y : num_\ell}\ \ell = \ell_x \sqcup \ell_y$$

$$\textsc{BoolArith}\ \frac{\Gamma \vdash x : bool_{\ell_x} \qquad \Gamma \vdash y : bool_{\ell_y}}{\Gamma \vdash x \diamond y : bool_\ell}\ \ell = \ell_x \sqcup \ell_y$$

$$\textsc{StrAdd}\ \frac{\Gamma \vdash x : str_{\ell_x} \qquad \Gamma \vdash y : str_{\ell_y}}{\Gamma \vdash x + y : str_\ell}\ \ell = \ell_x \sqcup \ell_y$$

$$\textsc{StrMul}\ \frac{\Gamma \vdash s : str_{\ell_s} \qquad \Gamma \vdash n : num_{\ell_n}}{\Gamma \vdash s * n : str_\ell}\ \ell = \ell_s \sqcup \ell_n$$

$$\textsc{Len}\ \frac{\Gamma \vdash s : string_\ell}{\Gamma \vdash len(s) : num_\ell} \qquad \textsc{Index}\ \frac{\Gamma \vdash s : str_\ell \qquad \Gamma \vdash n : int_\bot}{\Gamma \vdash s[n] : string_\ell} \qquad \textsc{Slice}\ \frac{\Gamma \vdash s : string_\ell \qquad \Gamma \vdash n_i : int_\bot}{\Gamma \vdash s[n_1 : n_2] : string_\ell}$$

Commands.

$$\textsc{Assn}\ \frac{\Gamma \vdash e : \tau_\ell \qquad pc \sqsubseteq \Gamma(x)}{\Gamma, pc \vdash x := e \Downarrow \Gamma[x \mapsto \tau_\ell]}$$

$$\textsc{Seq}\ \frac{\Gamma, pc \vdash c_1 \Downarrow \Gamma' \qquad \Gamma', pc \vdash c_2 \Downarrow \Gamma''}{\Gamma, pc \vdash c_1; c_2 \Downarrow \Gamma''} \qquad \textsc{Skip}\ \frac{}{\Gamma, pc \vdash \textbf{skip} \Downarrow \Gamma}$$

$$\textsc{If}\ \frac{\Gamma \vdash b : bool_\ell \qquad \Gamma, pc \sqcup \ell \vdash c_i \Downarrow \Gamma'}{\Gamma, pc \vdash \text{if } b \text{ then } c_1 \text{ else } c_2 \Downarrow \Gamma'} \qquad \textsc{While}\ \frac{\Gamma \vdash b : bool_\ell \qquad \Gamma, pc \sqcup \ell \vdash c \Downarrow \Gamma'}{\Gamma, pc \vdash \text{while } b \text{ do } c \Downarrow \Gamma'}$$

3.3 Type System

3.3.1 Types. The foundation of our secure types is the `SecureType` class, which is an abstract base class that all our custom objects inherit. Unlike more traditional object-oriented languages, Python does not have interfaces, so an abstract base class is the closest construct we can use to ensure that all the classes we are interested in share common methods. In addition, since it is abstract, the user cannot actually initialize a `SecureType` object. Any class that inherits from `SecureType` must define methods to get and set the value it holds and the security level, as well as the `__str__` and `__repr__` methods that convert the value held to a string and provides a string representation of the object respectively.

As for the actual secure types, we introduced three, all of which inherit `SecureType`:

- `SecureNum`. This type represents any numeric value (both integers and floats). It supports arithmetic and bitwise operations. Moreover, `SecureNum` objects can be compared using `<`, `>`, `<=`, `>=`, `==`, and `!=` operators; the result of this comparison is `SecureBool`.
- `SecureString`. This type represents a string object. It supports only a subset of `str` methods: concatenation, multiplication by a number (including `SecureNum`), indexing, slicing, and comparisons.
- `SecureBool`. This type represents boolean values. It supports logical operations: `&` (logical and), `|` (logical or), and `^` (logical xor).

3.3.2 Security Levels. In PIF, a security level l is defined such that $l \in \mathbb{N}$, thus allowing any arbitrary, non-negative integer to specify a security level. The information flow policy of PIF allows for flow from one object 1 to another object 2 if $l_1 \leq l_2$; thus, a higher security level is equivalent to higher clearance. An object may be reassigned to a higher security level, provided the current PC is less than or equal to the current security level of the variable to be assigned.

3.3.3 Program Counter. At any moment of the program's execution, the Program Counter (PC) indicates the security level of all information that influenced the fact that the current piece of code is being executed. For example, when the program is executing instructions inside an if statement, the PC security level should be at least the security level of the condition of that if statement. Modifying an object is allowed only if its security level is at least the level of the current PC

4 IMPLEMENTATION

4.1 Overview

The implementation[3] of our security-typed system alters code written by the user at the abstract syntax tree level. Our system relies on the usage of security labels to specify the relative clearance of an object. This is achieved via the `SecureNum`, `SecureString`, and `SecureBool` classes, in which each object has an integer type security field that specifies the relative security of the object and a value field that specifies the actual value of the object. In order to allow for standard integer, float, string, and bool operations, we defined the relevant magic functions for each class, which are special functions surrounded by two underscores on either side and which allow operator overloading. For example, if a `SecureNum` object with security level 1 and value 2 is added to a `SecureNum` object with security level 2 and value 2, the resulting object will be a `SecureNum` with security level 2 and value 4. Such behavior was achieved by defining the `__add__()` function for the `SecureNum` class.

4.2 Program Counter

In order to ensure correct information flow and handling of security types, we use a stack program counter that operations are pushed and popped from. An update to the program counter occurs when a new expression in the code is evaluated; the security level of the expression is considered against that of the stack and the expression is pushed to the program counter. The security level of the top of the stack is set to the maximum of the security levels of the incumbent top and the new evaluated expression. In order to support the pushing and popping of expressions, we use a class called `SecruityMonitor` to manage the stack with operations such as `push_pc(expr)`, `get_pc()`, and `pop_pc()`.

4.3 Python Implementation

In order to parse the PIF code written by a user, we use the Python `ast` module, which allows our application to process trees of the Python abstract syntax grammar and alter it accordingly. In the code, we control visiting nodes in the tree (assignments, if statements, etc.) and generate the appropriate checks of security level and type to ensure proper information flow between secure objects. The `ast` module then generates a separate Python file that features the underlying handling of secure values. This file is then executed with a standard Python interpreter while ensuring the expected security guarantees.

4.4 If statements

Any if statement affects the control flow of the program – some pieces of code are executed only depending on some condition. This creates a necessity for monitoring information flow changes created by if statements.

This is implemented the following way: any condition passed inside an if statement can have some security level: for example, if `a = SecureNum(5, 2)` and `b = SecureNum(7, 3)`, then the test in the condition `if a > b` has security level 3. Therefore, execution of any code inside the if statement is dependent on some information with security 3. This means that before executing this statement, we need to push 3 on the pc stack. This value is popped from the stack after the if statement body ends. Therefore, the ast node transformer transforms any if statement into the sequence of operations shown in Figure 1.

```
temp = SecurityMonitor.get_sec('''<initial condition>''')
SecurityMonitor.push_pc(temp)
if(temp.get_value()):
    '''<initial if body>'''
else:
    '''<initial else>'''
SecurityMonitor.pop_pc()
```

Fig. 1. If statement implementation

Two things are worth noting: first, the initial condition is wrapped into `SecurityMonitor.get_sec` wrapper function to make sure that it is a `SecureType` variable; second, the actual if condition is `temp.get_value()`, since we want to test the actual value of the `temp` variable. Also, since the execution of any `elif` or `else` blocks is also dependent on the initial condition, then `temp` is popped from the pc stack only after all these blocks.

4.5 While Loops

Similarly to if statements, while loops also modify the control flow. The design of while loop implementation is similar to if statements (see Figure 2).

```
temp = SecurityMonitor.get_sec('''<initial condition>''')
SecurityMonitor.push_pc(temp)
while(temp.get_value()):
    '''<initial while body>'''
    SecurityMonitor.pop_pc()
    temp = SecurityMonitor.get_sec('''<initial condition>''')
    SecurityMonitor.push_pc(temp)
SecurityMonitor.pop_pc()
```

Fig. 2. While statement implementation

The main difference from if statements is that the test should be updated on each iteration of the loop. This means that the security level of the test might be changed during the loop execution, and these changes should be accounted for. This requires popping the old test level from the pc stack at the end of each iteration, then updating the value of `temp`, and finally pushing the new value on the pc stack once again. While loops can also be followed by an `else` block – similarly to if statements, the final condition is popped from PC stack only after this block.

4.6 Assignments

The implementation of if and while statements allow the `SecurityMonitor` to correctly maintain the current pc security level. The only places where security constraints can potentially be violated are assignments, which means that the AST node transformers should add specific checks whenever the assignment happens.

First of all, it is needed to check whether the considered assignment is assigning a new value to an existing variable, or declaring a new variable. This is done by checking Python built-in `locals()` and `globals()` dictionaries. After that, if the variable existed before the assignment, the function `SecurityMonitor.check_assignment(var)` is called to check whether the variable is allowed for assignment at this pc value – the variable can only be assigned if its security level is at least the level of current PC. If this constraint is violated, then `check_assignment` throws an error; otherwise, we perform the actual assignment (see Figure 3).

```
if 'a' in locals():
    SecurityMonitor.check_assignment(locals()['a'])
elif 'a' in globals():
    SecurityMonitor.check_assignment(globals()['a'])

a = SecurityMonitor.sec_type(SecureNum(5, 3))
```

Fig. 3. Assignment implementation

```python
from pif.stypes.secure_num import SecureNum

a = SecureNum(5, 3)
b = SecureNum(10, 8)

if a > 3:
    b = a + b

print(repr(b))
```

Fig. 4. User PIF Code

```python
from pif.runtime.security_monitor import SecurityMonitor
from pif.stypes.secure_num import SecureNum

if 'a' in locals():
    SecurityMonitor.check_assignment(locals()['a'])
elif 'a' in globals():
    SecurityMonitor.check_assignment(globals()['a'])
a = SecurityMonitor.sec_type(SecureNum(5, 3))

if 'b' in locals():
    SecurityMonitor.check_assignment(locals()['b'])
elif 'b' in globals():
    SecurityMonitor.check_assignment(globals()['b'])
b = SecurityMonitor.sec_type(SecureNum(10, 8))

temp = SecurityMonitor.sec_type(a > 3)
SecurityMonitor.push_pc(temp)
if temp.get_value():
    if 'b' in locals():
        SecurityMonitor.check_assignment(locals()['b'])
    elif 'b' in globals():
        SecurityMonitor.check_assignment(globals()['b'])
    b = SecurityMonitor.sec_type(a + b)
SecurityMonitor.pop_pc()
print(repr(b))
```

Fig. 5. Generated Python Code

4.7 Well-Typed Example

To explain in further depth how PIF performs, we will look at the example shown in Figure 4. The code is well-typed and respects the PIF language specifications. We can see that `a` is declared as a `SecureNum` with value 5 and security level 3, while `b` is declared as a `SecureNum` with value 10 and security level 8. An if statement then checks if `a > 3`, and sets `b` equal to `a + b` if this is true. The resulting output of the program, not shown here is `'<SecureNum: val: 15, sec:8>'`; because the security level of `a` is less than that of `b`, information about `a` can flow to `b` and PIF allows updating the value of `b`.

Figure 5 shows the PIF-generated code from the user-written code. First, we can see that when declaring variable `a`, we first check whether `a` already exists; if so `check_assignment()` is called to ensure that `a` is not already a `SecureNum` that is being rewritten to an object with a lower security level. The same is repeated for the declaration of `b`. Then, a temporary variable, `temp`, is declared as the `SecureBool` expression `a > 3` in the if statement, and the variable is pushed to the program stack. We call `get_value()` on this `temp` boolean to check whether it is true or false, and `a + b` is written to `b` while checking that the security level of the if statement as well as variable `a` is less than that of `b`. The expression is then popped off the stack and the result is printed.

4.8 Ill-Typed Example

```python
from pif.stypes.secure_num import SecureNum

a = SecureNum(5, 3)
b = SecureNum(10, 8)

if b > 5:
    a = 10

print(repr(a))
```

Fig. 6. User PIF Code

```python
from pif.runtime.security_monitor import SecurityMonitor
from pif.stypes.secure_num import SecureNum

if 'a' in locals():
    SecurityMonitor.check_assignment(locals()['a'])
elif 'a' in globals():
    SecurityMonitor.check_assignment(globals()['a'])
a = SecurityMonitor.sec_type(SecureNum(5, 3))

if 'b' in locals():
    SecurityMonitor.check_assignment(locals()['b'])
elif 'b' in globals():
    SecurityMonitor.check_assignment(globals()['b'])
b = SecurityMonitor.sec_type(SecureNum(10, 8))

temp = SecurityMonitor.sec_type(b > 5)
SecurityMonitor.push_pc(temp)
if temp.get_value():
    if 'a' in locals():
        SecurityMonitor.check_assignment(locals()['a'])
    elif 'a' in globals():
        SecurityMonitor.check_assignment(globals()['a'])
    a = SecurityMonitor.sec_type(10)
SecurityMonitor.pop_pc()
print(repr(a))
```

Fig. 7. Generated Python Code

Now, let us look at an ill-typed example of user code in Figure 6. The code here violates information flow. We can see that `a` is declared as a `SecureNum` with value 5 and security level 3, while `b` is declared as a `SecureNum` with value 10 and security level 8. An if statement then checks if `b > 5`, and sets `a` equal to 10 if this is true. As information can not flow from `b` to `a`, PIF raises an error for this code: *pif.runtime.SecurityException: Security level of the expression is too low.*

The code in Figure 7 shows the PIF-generated code from the user-written code. Similar to the well-typed example we can see that we check if `a` and `b` already exist and declare them accordingly. Then, a temporary variable, `temp`, is declared as the boolean expression `b > 5` in the if statement, and the variable is again pushed to the program stack. Unlike the well-typed example, this now throws an error. After writing the if statement with security level 8 to the stack, any assignment must be at security level 8 or higher. As such, when the PIF-generated code attempts to set `a` to 10, an error is thrown as the security level of `a` is not high enough and reveals information about `b` to low security parties.

5 CHALLENGES & FIXES

5.1 Commutativity of Binary Operations

The implementation of our Python library and `SecureNum` class addresses arithmetic, bitwise, and comparison operands by overriding the operand functions to ensure that we can treat our secure types as integers and can perform operations.

The original implementation of our Python library only overrode left operands (ie, `__add__`, `__sub__`, `__mul__`, etc.). As such, any operations with a `SecureNum` on the right side of the operation character, such as `3 + b`, would throw an error. Along with each of the standard operands, we also had to override each of the corresponding right operands for these (ie, `__radd__`, `__rsub__`, `__rmul__`, etc.) to prevent these errors from occurring.

5.2 Assignments vs Reassignments

Python allows for arbitrary reassignment of variables; for example if we have a variable `a = 5`, we can reassign the value of `a` to `a = [1,2,3]`, `a = "Hi"`, or `a = func()`. This presents a challenge for our security-typed language, as Python allows for the reassignment of secure objects to another secure object of lower security level, or an object such as a string or list that is not supported by our language. In order to prevent this problem from occurring, we incorporate a check in the `globals()` and `locals()` dictionaries to track current variable assignment and to ensure that we are not incorrectly reassigning a secure object to an unsupported value.

5.3 SecurityMonitor Circular Dependency

Initially, we designed our code in such a way that `SecureNum` relied on `SecurityMonitor` and vice versa, which created a circular dependency and led to the code being (1) hard to divide and therefore maintain and (2) hard to test. For example, `SecurityMonitor` had functions to extract the value and level of a `SecureNum` and `SecureNum` relied on the `SecurityMonitor` class to wrap non-secure objects in a secure type. We resolved this by introducing the `SecureType` abstract base class from which all our secure classes inherit. This ensured that all secure types had the same functions defined, such as `get_value()` and `get_level()`, which delegate getting values and levels to the classes rather than `SecurityMonitor`. Furthermore, each class has its own `wrapper()` function which is static and responsible for wrapping a non-secure type in a security container. For example, `SecureNum.wrapper` function takes an `int` or `float` value and returns a `SecureNum` object.

5.4 Strings

Strings in Python have many methods, and `SecureString` implementation supports only a subset of them since it is difficult to reason about the security of all these methods. Nevertheless, `SecureString` supports indexing, slicing, concatenation, multiplication by `SecureNum` (that concatenates the string with itself several times), and getting the length of the string.

5.4.1 Length. There are challenges related to getting a string's length. Although it is possible to define a built-in `len()` function for any class (including `SecureString`), the Python specification requires `len()` to return a value of type `int`. This would violate security, since the length of a string reveals some information about the string, meaning that it should have a security level. Therefore, when working with `SecureString` objects, it is required to use a custom `.get_length()` method, which returns a `SecureNum`.

5.4.2 Out of Bounds Errors. For `SecureString`, getting an index/slice returns a `SecureString` with a security level matching the level of the initial string. Thus, indexing does not leak any secure information. However, the default implementation of indexing and slicing for Python strings throws an error when the requested indices are out of bounds – observing such an error would leak information about the length of a string. To prevent that, `SecureString` behaves differently – when the requested index is out of bounds, it still returns a `SecureString` object, but its value is just an empty string. This way, no errors could be thrown, meaning that there are no information leaks due to certain exceptions.

5.5 Booleans: and, or

In Python, it is impossible to overload `and` and `or` operations, since they both apply only to `bool` objects (their operands are always cast to `bool` first). Therefore, instead of `and` and `or`, we use operators `&` and `|` to work with `SecureBool` objects.

5.6 Try-Except

Existing work such as Jif[1] leverage the integration of exceptions in the type system of Java. This lets the compiler know when an exception may occur due to a secure value. In Python, we have much simpler support for exceptions. In order to handle the `try-except` control flow in a security-type context, we require a more course-grained approach. Consider the least-upper-bound of the security levels of each function argument, along with each possibly failing expression such as `s[n]` in the `try` block. This least-upper-bound would become the `pc` in the `except` block, where our existing semantics would show that the `except` block is well-typed. The intuition is that control flow shifts to the `except` block if something in the `try` block causes an exception to be thrown, thus the course-grained approach would set `pc` as described above. With our implementation, this could be possible with the reference monitor, but would require significant reworking of our formal semantics, which leaves it out of scope.

6 SECURITY GUARANTEES

PIF uses a dynamic reference monitor to enforce security typing in a subset of the Python programming language. It guarantees dynamic, flow-sensitive information flow enforcement. Flow-sensitivity allows variables' security levels to change throughout the program. We follow the no-sensitive upgrade condition[4] to combat information leakage by the decision to upgrade a variable's security level. It requires that $pc \sqsubseteq \Gamma(x)$, intuitively, the level of the decision to upgrade (determined by the program counter) must flow to the current security level of the variable to be assigned. Our implementation does not attempt to prevent information leakage through channels such as timing or termination. Additionally, as discussed further below, use of other Python language features outside of the scope of our implementation may break the security guarantees we have outlined.

7 FUTURE WORK AND LIMITATIONS

Due to the array of baked-in Python functions for different objects, we limited our projects focus to support integers, floats, bools, and strings. Implementing security-typed support for elements such as lists requires the overloading and handling of built-in functions that are specific to these object types, such as `append()` and `pop()` for lists. Due to the large scope of this challenge, we leave such an implementation supporting these additional objects to future work.

Due to some of the Python-specific features such as function decorators, it is additionally difficult to ensure security of functions when dealing with decorated functions. As such, ensuring careful handling of these features will additionally be left to future work.

Besides, secure multi-level I/O needs to be added to ensure user programs can actually interact with the outside world. This would require adding methods to set up I/O channels and specify the security levels that they support.

So far, PIF supports information control for single Python files – the user cannot import any other PIF files or use certain Python standard library modules. Adding support for the Python Standard Library and allowing users to connect multiple PIF files into a single system is a step that is essential to allow users to build large real-life projects.

The security label system is now extremely simple – security levels are represented as non-negative integers. In many cases, a more complex label system might be required. Therefore, we need to add support for arbitrary security label lattices to make the label system more flexible.

Finally, some of Python's syntactic sugar that's not implemented at this point could be added. This includes unpacking / multi-assignment of the form `a, b = b, a` and list comprehensions (`[-x for x in range(10)]`). This would require extending the functionality of PIF's AST node transformer.

8 CONCLUSION

In this paper, we presented PIF, a new security-typed language that can be used to ensure proper information flow and protocol modeling. Similar to the Jif programming language, labels of security levels are assigned to secure objects. PIF ensures that the operations for each of these is permitted given the security levels and will raise an error if the information flow is ill-formed. As discussed in the future work and limitations section, there is a lot of potential to extend our language to support additional features such as lists as well as Python-specific constructs such as decorator functions.

REFERENCES

[1] JIF (2016), https://www.cs.cornell.edu/jif/
[2] Jeeves (2013), https://projects.csail.mit.edu/jeeves/about.php
[3] PIF (2024), https://github.com/l3cire/python-information-flow/tree/main
[4] Austin, Flanagan (2009), *Efficient Purely-Dynamic Information Flow Analysis*, PLAS '09